JNBridgePro™ Users' Guide

Version 12.0

jnbridge.com

JNBridge, LLC
jnbridge.com

April 7, 2025

Table of Contents

How to use this guide

This guide contains information about installing, configuring, and using JNBridgePro. It is organized according to the particular task or tasks you wish to perform with JNBridgePro.

- ***All users*** are encouraged to read the **"JNBridgePro overview"** section below. It gives information on the architecture of JNBridgePro, and the various scenarios in which it can be used. It lists the components that come with the installation, and discusses the various communication mechanisms between the .NET and Java sides, including shared-memory communications.
- Users who have ***purchased licenses*** or are ***planning a deployment*** of JNBridgePro should read the **"Licensing"** section, which describes the various licensing mechanisms, and discusses the various ways of deploying licenses.
- If you're creating a ***.NET-calling-Java*** project, read the section **"Using JNBridgePro for .NET-to-Java calls.**" This section has four subsections:
  - **"Generating proxies with JNBridgePro"** describes how to generate the .NET-side proxy classes through which Java classes are accessed from .NET.
  - **"Using Proxies with JNBridgePro"** describes how to use JNBridgePro to achieve .NET/Java interop in the context of the application you are developing. This section discusses the use of proxies and JNBridgePro runtime components, how JNBridgePro is configured, and how it is used in various scenarios, including inside Java EE application servers, and as part of an ASP.NET Web application, among other situations. It also describes how to tune network performance and how to override the default garbage collection behavior for more precise memory management.
  - **"Mapping Java entities to .NET"** describes the ways in which Java classes are mapped to their corresponding proxies. It also discusses how callbacks are created and used, and the difference between by-reference and by-value proxies.
  - **"Transaction-enabled classes and support for transactions"** describes how multiple calls from .NET to Java can participate in the same Java transaction.
- If you're creating a ***Java-calling-.NET*** project, read the section **"Using JNBridgePro for Java-to-.NET calls."** This section is very similar to the previous one, except that it focuses on projects where Java code calls .NET code. This section has four subsections:
  - **"Generating proxies with JNBridgePro"** describes how to generate the Java-side proxy classes through which .NET classes are accessed from Java.
  - "**Using Proxies with JNBridgePro"** describes how to use JNBridgePro to achieve .NET/Java interop in the context of the application you are developing. This section discusses the use of proxies and JNBridgePro runtime components, how JNBridgePro is configured, and how it is used in various scenarios. It also describes how to tune network performance and how to override the default garbage collection behavior for more precise memory management.
  - **"Mapping .NET entities to Java"** describes the ways in which .NET classes are mapped to their corresponding proxies. It also discusses how callbacks are created and used, and the difference between by-reference and by-value proxies.

    - **"Transaction-enabled classes and support for transactions"** describes how multiple calls from Java to .NET can participate in the same .NET transaction.

- If you're using JNBridgePro with .NET Core, please see the sections marked ".NET Core" for instructions on how JNBridgePro for .NET Core differs from JNBridgePro on the traditional .NET Framework platform. In addition, these .NET Core-specific sections are collected in a document *JNBridgePro for .NET Core Users' Guide*. For any issue not covered in the .NET Core-specific materials, please see the other relevant parts of this *Users' Guide*, as these aspects of the .NET Core-targeted version should be identical to those of the .NET Framework-targeted version.
- If you're creating a ***bi-directional*** project, you should read the above two sections, plus the section **"Use of JNBridgePro for bi-directional interoperability."** This will give you the additional information you will need to have calls going in the .NET-to-Java and Java-to-.NET directions in the same program.
- If you are ***upgrading an application using an old version of JNBridgePro***, you may want to see the appendix on the jnbproxy.config file. Earlier versions of JNBridgePro were configured through the special jnbproxy.config file, although this is no longer necessary, and is in fact discouraged.

Please also see the *JNBridgePro Release Notes* for the latest information on system requirements and known issues.

JNBridgePro overview

JNBridgePro is a bi-directional Java/.NET interoperability tool that enables you to connect your Java-based and .NET-based components and APIs together. Java code can call .NET code that is written in .NET-based languages, and Java classes can be extended by having classes that are written in other languages inherit from the Java classes. .NET code can also call Java code, and .NET classes can be written that extend Java classes. JNBridgePro achieves cross-language interoperability as it retains Java's cross-platform capability and conformance to Java standards. The Java classes called by .NET are in the form of Java bytecodes; Java source code need not be available. Similarly, the .NET classes called by Java are in the form of MSIL (Microsoft Intermediate Language); .NET source code need not be available.

Architecture of JNBridgePro

When JNBridgePro is used to interoperate between .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). The CLR and the JVM may run on the same machine, or on different machines connected by a network. Every application using JNBridgePro consists of one or more instances of the CLR, and one or more instances of the JVM. In this document, the set of all .NET CLRs in such an application will be collectively referred to as the ".NET-side," and the set of all instances of the JVM will be referred to as the "Java-side."

JNBridgePro supports communication between .NET and Java code using two communications mechanisms: a binary/TCP protocol, or shared-memory communications. Shared-memory communications runs the Java virtual machine in the same process as the .NET client, and does not use sockets as the basis for communications, as do the other two mechanisms. The binary protocol is very fast (although not as fast as shared memory), but may not work across some firewalls. (*Prior to version 10.1, an HTTP/SOAP communications mechanism was also offered. This mechanism has been removed.*)

This architecture is highlighted in Figure 1below.

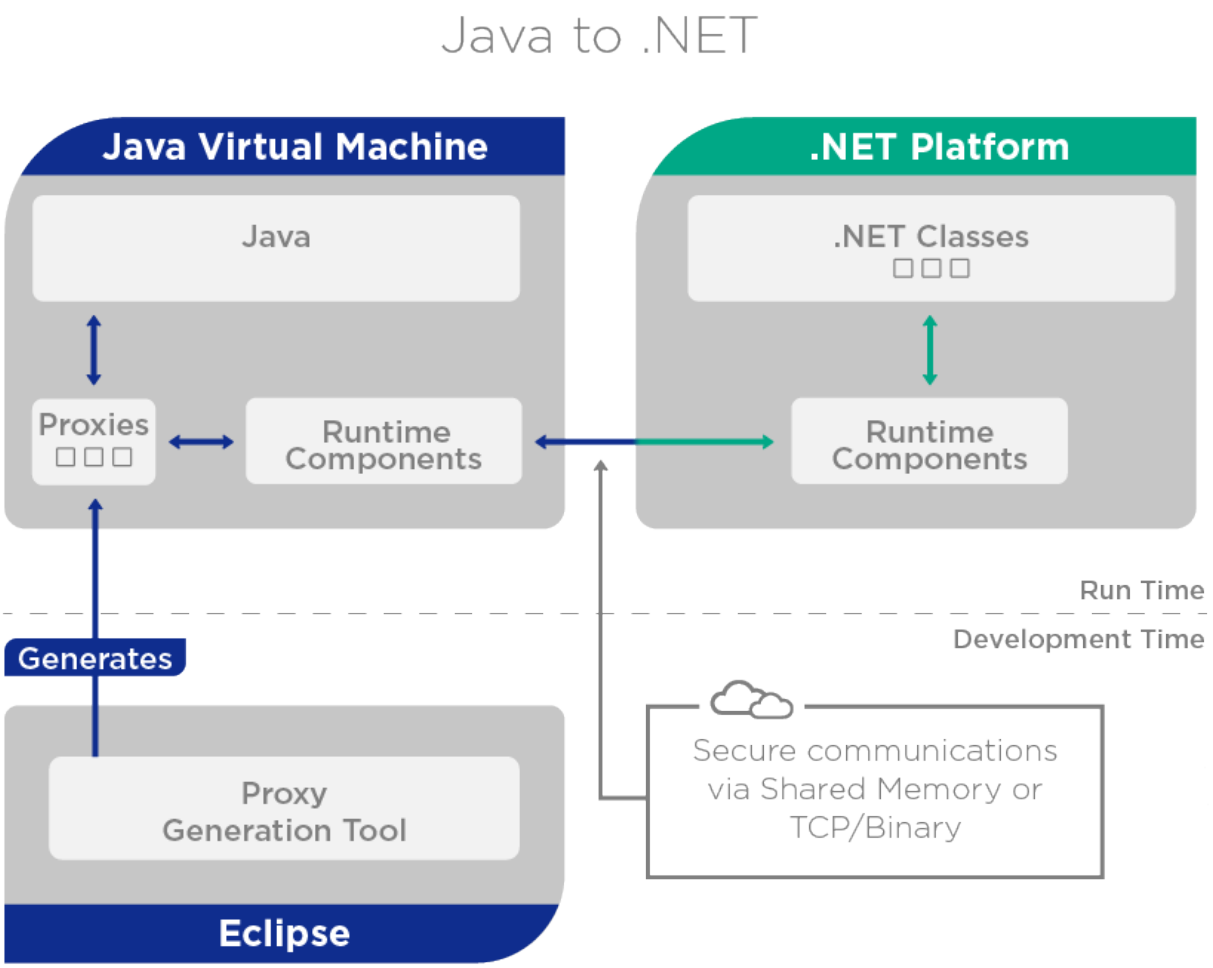
Usage scenarios

Direction: Java-calling-.NET versus .NET-calling-Java

JNBridgePro usage scenarios can be classified in two different ways. First, the scenarios may be classified according to *direction*. JNBridgePro may be used in either a *.NET-calling-Java* direction, or in a *Java-calling-.NET* direction. In the .NET-calling-Java direction, code written in a .NET language, and running on the .NET platform, accesses Java code running on the Java platform. The access is done through proxies on the .NET side that correspond to underlying Java classes and objects, and which manage communication with those underlying classes and objects. In the Java-calling-.NET direction, code written in Java, and running on the Java platform, accesses code running on the .NET platform. The Java-to-.NET access is done through Java-side proxies that correspond to underlying .NET classes and objects. Systems using JNBridgePro can use it in both directions simultaneously, so that a system can have code with both calls from .NET to Java and calls from Java to .NET; special considerations when performing such *bi-directional interoperation* are given at the end of this document.

Proxy Generation versus Proxy Use

Second, JNBridgePro usage can be classified as either *proxy generation* or *proxy usage* scenarios. In a *proxy generation* scenario, typically during development, the developer specifies the underlying classes (Java or .NET, depending on the direction) with which the developer's classes on the other platform will communicate, and then generates the corresponding proxy classes. In a *proxy use* scenario, typically during testing and production, classes on one platform (either Java or .NET, depending on the direction) communicate with classes on the other platform by way of the generated proxy classes. JNBridgePro is installed, configured, and used in a slightly different manner for each scenario. Usage differences are described in subsequent sections of this manual.



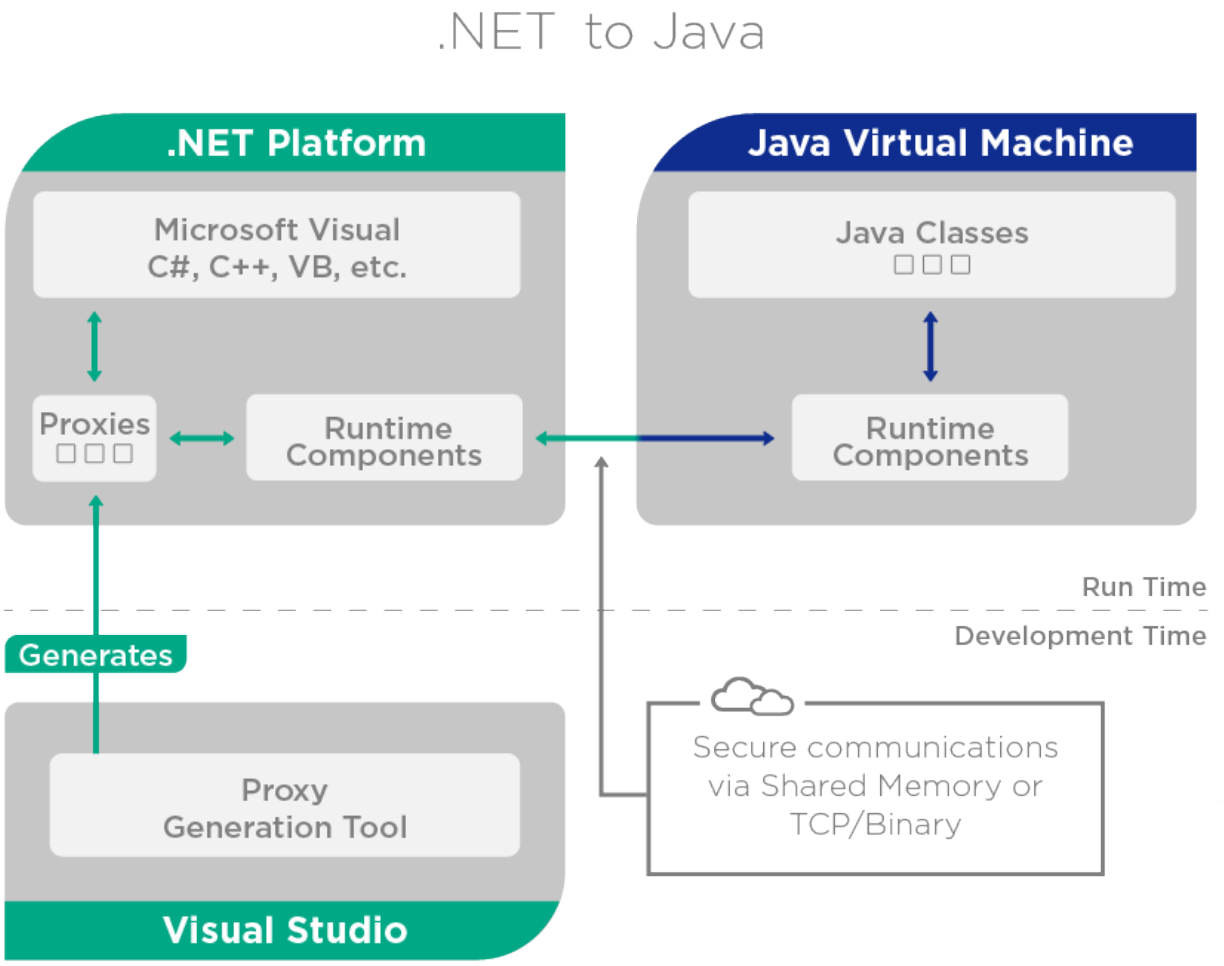


Figure 1. JNBridgePro architecture.

Shared-memory communications

JNBridgePro supports both socket-based and shared-memory communications. The binary protocol is socket-based, which means that communications is performed through sockets even when the .NET side and the Java side are running in the same process. Although great effort has been made to optimize performance, socket-based communication still incurs some overhead. In addition, the two socket-based mechanisms typically require the user to explicitly start and stop the Java side.

JNBridgePro also supports shared-memory communication, in which the Java virtual machine is automatically started up in the same process as the .NET client, and communication between the .NET and Java sides is through shared data structures; no socket overhead is incurred. Performance improvements of between 25 and 50 percent may be observed.

The following restrictions apply to shared-memory communication between .NET and Java:

- The .NET and Java sides must reside on the same machine. If the .NET and Java sides reside on different machines, socket-based communication must be used.

- The Java side must run inside a standard JVM. Java EE application servers cannot be communicated with through shared-memory communication; use socket-based () communication in such cases.
- The Java side can only communicate with one .NET side: the .NET side in whose process the JVM is running. If the Java side must communicate with more than one .NET side, use socket-based communication.

In order for Java-to-.NET calls to use shared-memory communication, the files jnbshare.dll, jnbsharedmem.dll (actually, either jnbsharedmem_x86.dll, jnbsharedmem_x64.dll, or both), and jnbjavaentry2.dll (actually, jnbjavaentry2_x86.dll, jnbjavaentry2_x64.dll, or both) must all be installed in the Global Assembly Cache (GAC), or in the folder specified by the dotNetSide.appBase property (see "Specifying the .NET-side application base folder," below). This includes situations where bidirectional shared-memory communication is used. See the .NET documentation, particularly the documentation on gacutil.exe, for information on how to install an assembly in the GAC. ***If you are not performing Java-to-.NET calls using shared memory, it is not necessary to install these assemblies in the GAC or in the dotNetSide.appBase folder.***

When using .NET Core: When using shared memory with .NET Core, use the following components:

- .NET Core-to-Java:
  - jnbshare.dll, plus jnbshare.deps.json and required extension dlls (see below)
  - jnbsharedmem_x86.dll or jnbsharedmem_x64.dll (if Windows), or libJNBSharedMem_x64.so (if Linux)
- Java-to-.NET Core:
  - jnbshare.dll, plus jnbshare.deps.json and required extension dlls (see below)
  - jnbsharedmem_x86.dll or jnbsharedmem_x64.dll (if Windows), or libJNBSharedMem_x64.so (if Linux)
  - jnbjavaentry_x86.dll or jnbjavaentry_x64.dll (if Windows), or libJNBJavaEntry_x64.so (if Linux)
  - jnbjavaentry2.dll

JNBridgePro components

JNBridgePro consists of a number of components, along with associated configuration files. These are:

- **JNBProxy GUI version** (jnbproxygui.exe and jnbproxygui32.exe) is the tool used to generate the .NET proxies through which other .NET classes communicate with Java classes. jnbproxygui32.exe is a 32-bit version of the proxy generation tool. It is only installed on 64-bit systems, and should only be used when generating proxies for Java-to-.NET projects and the DLLs containing the classes to be proxied are 32-bit. For all other situations, use jnbproxygui.exe, which will run as a 64-bit application on 64-bit systems, and as a 32-bit application on 32-bit systems.
- **JNBProxy command-line version** (jnbproxy.exe and jnbproxy32.exe) is a command-line version of the proxy generation tool. jnbproxy32.exe is a 32-bit version of the proxy generation tool. It is only installed on 64-bit systems, and should only be used when generating proxies for

Java-to-.NET projects and the DLLs containing the classes to be proxied are 32-bit. For all other situations, use jnbproxy.exe, which will run as a 64-bit application on 64-bit systems, and as a 32-bit application on 32-bit systems.

- **JNBProxy Visual Studio plug-in** (jnbridgeplugin.dll and jnbridgeplugin.11.dll) is a version of the proxy generation tool that works as a plug-in for Visual Studio 2005, 2008, 2010, 2012, 2013, and 2015.
- **JNBProxy Visual Studio plug-in** (jnbridgeplugin2017.dll and jnbridgeplugin2017.vsix) is a version of the proxy generation tool that works as a plug-in for Visual Studio 2017, 2019 and 2022. See "Installing the Visual Studio plugin (Visual Studio 2017, 2019 or 2022)" for more details.
- **JNBProxy Eclipse plug-in** (jnbridgepro11_0_0-eclipse.zip) is a version of the proxy generation tool that works as a plug-in for Eclipse 3.2 through 4.10.
- **JNBPCommon** (jnbpcommon.dll) contains functionality shared by the GUI and command-line versions of JNBProxy.
- **JNBShare** (jnbshare.dll) is a .NET assembly that contains core functionality used by all generated .NET proxies.
- **JNBSharedMem** (jnbsharedmem_x86.dll and jnbsharedmem_x64.dll) is a .NET assembly (containing Win32 or x64 code, respectively) that contains functionality used by the shared-memory communications channel. Use jnbsharedmem _x86.dll if you are running a 32-bit process, jnbsharedmem _x64.dll if you are running a 64-bit process, and include both if your application can be run as either a 32-bit or 64-bit process (i.e., an "Any CPU" application).
- **JNBWPFEmbedding** (jnbwpfembedding.dll) is a .NET assembly that contains functionality used when embedding Windows Presentation Foundation (WPF) components inside Java applications, and vice versa.
- **JNBJavaEntry** (jnbjavaentry_x86.dll and jnbjavaentry_x64.dll) and **JNBJavaEntry2** (jnbjavaentry2_x86.dll and jnbjavaentry_x64.dll) are a Win32 (or x64) library and a 32-bit or 64-bit .NET assembly, respectively, that contain functionality used by the shared-memory communications channel to support Java-to-.NET calls.
- **JNBDotNetSide** (jnbdotnetside.exe and jnbdotnetside32.exe) are executables that provide an easy way to run the .NET side in Java-to-.NET projects. It is not necessary to use these, as the APIs for starting and running the .NET side are available to be incorporated in your own code, but these provide a convenient way to do so in many cases. jnbdotnetside.exe runs as a 64-bit process on 64-bit systems, and as a 32-bit process on 32-bit systems, while jnbdotnetside32.exe always runs as a 32-bit process, regardless of platform.
- **JNBCore** (jnbcore.jar) is a Java jar file that manages communication between Java classes and .NET classes, as well as the object lifecycle (creation, use, and destruction) of Java objects created by calls from .NET objects.
- **Byte Code Engineering Library** (bcel-6.*n*.*m*.jar) is a Java jar file containing a version of the Apache Byte Code Engineering Library (BCEL), which contain functionality to generate new Java-side proxy classes. Versions of JNBridgePro prior to version 12.0 used a modified version of the BCEL library but this usage has been updated to use an unmodified version of BCEL, initially using version BCEL version 6.10.0.

- **Registration Tool** (RegistrationTool.exe) is used to register and license JNBridgePro on any machine on which you want JNBridgePro (either the installed JNBridgePro, or your application using the JNBridgePro .NET components – JNBShare, in particular) to run.
- **JNBAuth** (jnbauth_x86.dll, jnbauth_x64.dll, jnbauth_x86.so, jnbauth_x64.so , in previous versions: rlm932_x86.dll, rlm932_x64.dll, rlm932_x86.so, rlm932_x64.so) contain the license management functionality, and must be included in your application. Use jnbauth_x86.dll if you are running a 32-bit process, jnbauth_x64.dll if you are running a 64-bit process, and include both if your application can be run as either a 32-bit or 64-bit process. Use jnbauth_x86.so or jnbauth_x64.so when running the .NET-side on Linux. See "Licensing on Linux" for more details.
- **Ionic ZipLib** (Ionic.Zip.dll) is a library with the functionality to read the contents of Zip and Jar files. It is used with the JNBProxy GUI version to support class name completion.

During *proxy generation*, eight Development Components (JNBProxy [GUI, Visual Studio plug-in, Eclipse plug-in, or command-line], JNBPCommon, JNBShare, JNBSharedMem, Ionic ZipLib, JNBAuth, JNBCore, and BCEL) are required. During *proxy use*, only the Runtime Components JNBShare, JNBAuth and JNBCore are required (plus BCEL, if proxy use is in the Java-to-.NET direction, and JNBSharedMem, JNBJavaEntry, and JNBJavaEntry2, if shared-memory communication is being used), although the Registration Tool will be required if a license has not yet been installed on the machine.

4.8-targeted version

Starting with version 12.0 JNBridgePro supports .NET Framework 4.8. Earlier versions of JNBridgePro included two sets of binaries, one for .NET Framework 2.x and one for 4.x. The .NET Framework 2.x is no longer supported and the .NET Framework 4.0 product has been updated to .NET Framework 4.8 and the binaries are available in the "4.8-targeted" directory. As before there are variants for both 32-bit and 64-bit applications.

For historical reasons the 4.8-targeted version's components are labeled with the .NET file version 12.10.0.0.

JNBridgePro for .NET 8 (also called .NET Core)

Differences between .NET Framework and .NET Core versions of JNBridgePro

The following features of "traditional" JNBridgePro are not available in JNBridgePro for .NET Core:

- Failover.
- Embedding of Java UI elements in .NET Windows Forms and WPF applications
- Embedding of .NET Windows Forms and WPF elements in Java AWT, Swing, or SWT applications.

Components

JNBridgePro for .NET Core and .NET 8 includes a number of components. Note that applications are distributed as framework-dependent DLLs, not as framework-independent EXEs.

- jnbproxy.dll: The command-line proxy generation tool for generating proxy JAR files for Java-to-.NET Core projects. jnbpcommon.dll, which is also part of the "traditional" JNBridgePro proxy generation tool, is not needed in the .NET Core-targeted tool. Also, JNBProxy.deps.json and JNBProxy.runtimeconfig.json – both these files must be in the same folder as jnbproxy.dll.
- registrationTool.dll: A command-line version of the registration tool that can run on .NET Core. Also, registrationTool.deps.json and registrationTool.runtimeconfig.json – both these files must be in the same folder as registrationTool.dll.
- jnbdotnetside.dll: A command-line version of the .NET-side for Java-to-.NET Core projects that can run on .NET Core. Also, jnbdotnetside.deps.json and jnbdotnetside.runtimeconfig.json – both these files must be in the same folder as registrationTool.dll.
- jnbshare.dll: The central JNBridgePro .NET-side runtime component. It performs the same functions as the jnbshare.dll in the "traditional" JNBridgePro.

To run, jnbshare.dll depends on the following NuGet packages, which must be downloaded and included with your project:

- Microsoft.Extensions.Configuration.Abstractions
- Microsoft.Extensions.Configuration.Binder
- Microsoft.Extensions.Configuration.Json
- Newtonsoft.Json

- jnbsharedmem_x64.dll, jnbsharedmem_x86.dll, libJNBSharedMem_x64.so: The .NET-side runtime component containing functionality supporting shared memory communication, for 64-bit Windows, 32-bit Windows, and 64-bit Linux, respectively. Use the particular version as appropriate for your application. Alternatively, include all of them and the appropriate one will be used, depending on underlying platform and process bitness. Only 64-bit Linux is being supported.

- jnbjavaentry_x64.dll, jnbjavaentrry_x86.dll, and libJNBJavaEntry_x64.so: .NET-side runtime components containing functionality for shared memory in the Java-to-.NET Core direction, for 64-bit Windows, 32-bit Windows, and 64-bit Linux, respectively. Use the particular version as appropriate for your application. Alternatively, include all of them and the appropriate one will be used, depending on underlying platform and process bitness. Only 64-bit Linux is being supported.

- jnbjavaentry2.dll: .NET-side runtime components containing additional functionality for shared memory in the Java-to-.NET Core direction. It will work on all supported .NET Core platforms. Note that, unlike "traditional" JNBridgePro, there is only a single version of this assembly, not multiple versions depending on the bitness of the application's process.

- jnbauth_x64.dll, jnbauth_x86.dll, jnbauth_x64.so: Components containing licensing functionality for JNBridgePro, for 64-bit Windows, 32-bit Windows, and 64-bit Linux, respectively. Use the particular version as appropriate for your application. Alternatively, include all of them and the appropriate one will be used, depending on underlying platform and process bitness. Only 64-bit Linux is being supported.

All the above components are version 12.0.

Use the same Java-side components (jnbcore.jar and bcel-6.*n*.*m*.jar) that are used in "traditional" JNBridgePro.

Licensing

JNBridgePro must be licensed on each machine on which the file `jnbshare.dll` is installed. JNBridgePro supports a variety of license types. For more information, please visit https://jnbridge.com/software/jnbridgepro/license-and-purchase.

Note: If you have used versions of JNBridgePro older than version 6.0, note that starting with v6.0 we have introduced a new licensing mechanism. The concepts and components described below will be different from those you might have used in previous versions.

License files

For most types of licenses, the license is encapsulated in a *license file*, which is a text file whose suffix is *.lic*. (The file's name is generally assigned by JNBridge's license tracking mechanism.)

When a license file is present, the JNBridgePro tools, and your application that uses JNBridgePro, will look for the license file in the following locations, in order, until a license file is found:

- The location specified in the application's app.config file.
- The folder in which the application's executable file resides.
- In Java-to-.NET projects using shared memory, the application base folder specified using the dotNetSide.appBase property.
- If the JNBridgePro installer has been run on the machine, and development mode has been chosen, the JNBridgePro installation folder (for example, C:\Program Files (x86)\JNBridge\JNBridgePro v12.0).

If, at any time in the process, an invalid license file is found, an InvalidLicenseException is thrown and the search halts. A license file can be invalid for a number of reasons, including:

- It has been tampered with.
- It is time-limited and has expired.
- It is node-locked, and is being used on a machine other than the one to which the original license was locked.

Note: In Java-to-.NET projects, here are some guidelines for placement of the license file:

- When using shared memory, either place the license file in the application base folder specified using the dotNetSide.appBase property, or specify the license file's location in the app.config file using the licenseLocation element (see below), and use the dotNetSide.appConfig property to indicate the location of the app.config file.
- When using TCP/binary communications, either place the license file in the same folder as the .NET side's .exe file (usually either JNBDotNetSide.exe or a .NET side application that you have created), or specify the license file's location in the .NET-side .exe's app.config file using the licenseLocation element (see below).

Licensing and application configuration files

You have the option of specifying the location of your license file, or, if you are using a license server, the location (host and port) of the license server.

Inside the <configuration> section of the application configuration file, add the following section if it is not already there:

```
<configSections>
    <sectionGroup name="jnbridge">
            <section name="licenseLocation"
                    type="System.Configuration.SingleTagSectionHandler"/>
    </sectionGroup>
</configSections>
```

Inside the <jnbridge> section, add the following line:

<licenseLocation directory="*absolute or relative path to directory containing license file*"/>

or

<licenseLocation host="*license server machine*"
port="*license server port*"/>

Choose the first variant if you are using a license file; choose the second variant if you are using a license server. Again, specifying this information is optional. If you omit it, the licensing mechanism will continue looking for the license in the folder containing the application's executable file as described above.

Licensing components

The licensing mechanism resides in the files jnbauth_x86.dll and jnbauth_x64.dll. Use jnbauth_x86.dll if you are creating a 32-bit application. Use jnbauth_x64.dll if you are creating a 64-bit application. If you are creating an "Any CPU" application that can run as either a 32-bit or 64-bit process, use both jnbauth_x86.dll and jnbauth_x64.dll; the proper one will be chosen depending on whether your application is running as a 32-bit or 64-bit process. These files should be placed in the same folder as the copy of jnbshare.dll that is used by your application.

The JNBridgePro registration tool RegistrationTool.exe comes with the JNBridgePro installation and is used to manage licensing. If you are working on a machine on which the JNBridgePro installer has been run, you can access the registration tool through the Start menu. If you have deployed your application to a machine on which JNBridgePro has not been installed, you should copy RegistrationTool.exe over to your deployment machine and place it in the same folder as jnbshare.dll. *The registration tool must be run with administrative privileges.*

When you launch the registration tool, you will see a form similar to that shown in Figure 2. Registration Tool. Information on the JNBridgePro version and license will be displayed. Across the top are a number of tabs that are used for different licensing-related activities. Please see the sections below for various licensing-related scenarios and how to use the registration tool in those scenarios.

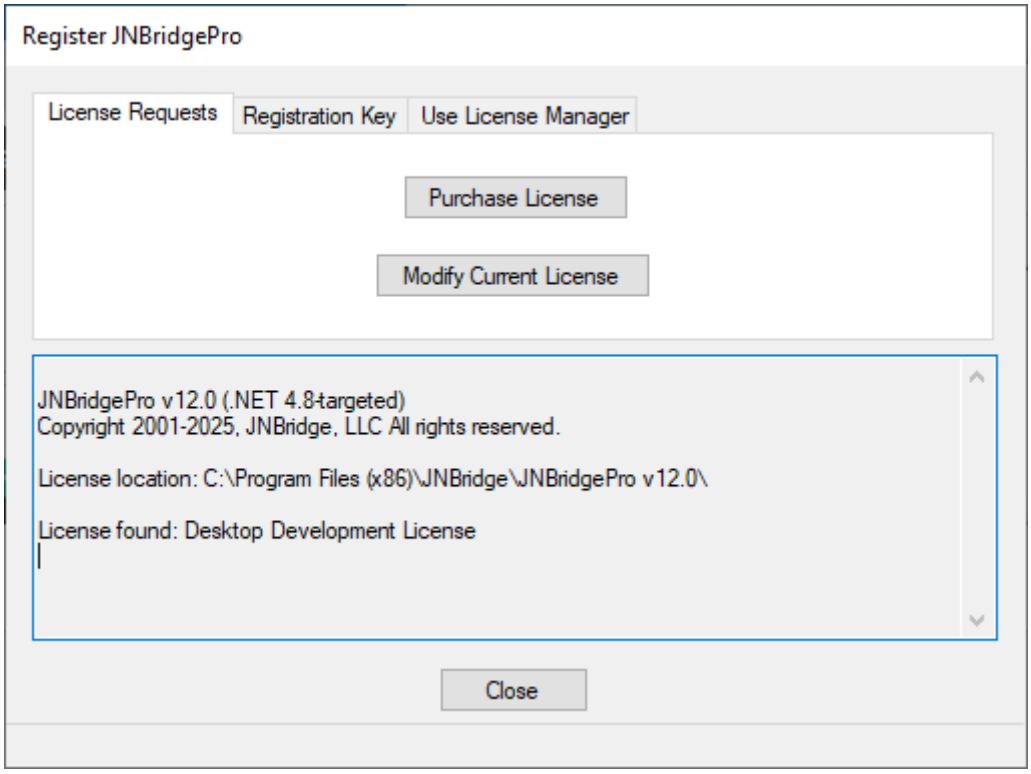


Figure 2. Registration Tool

Evaluation licenses

After installing JNBridgePro, use the Registration Tool as described below in the section "Obtaining licenses." You will be emailed a 30-day evaluation license. This will allow you to use the JNBridgePro product on a trial basis for 30 days. ***Note:*** you must perform this action before you can begin your evaluation.

The evaluation license file *is* node-locked, and cannot be copied to other machines. If do you wish to place evaluation installations of JNBridgePro on additional machines, you can request licenses using the registration tool on the new machines at any time and receive 30-day evaluation licenses on those machines. Please note that any deployments and installations using evaluation licenses will stop working after the 30-day evaluation period expires. Also note that, under the terms of your JNBridgePro license, you may not use your evaluation license to run production applications.

If you need an extension to your 30-day evaluation period, please visit https://jnbridge.com/support/license-key and follow the instructions there, or contact registration@jnbridge.com.

Obtaining licenses

You can request an evaluation license or claim a permanent license by launching the registration tool, then selecting the "Registration Key" tab (Figure 3).

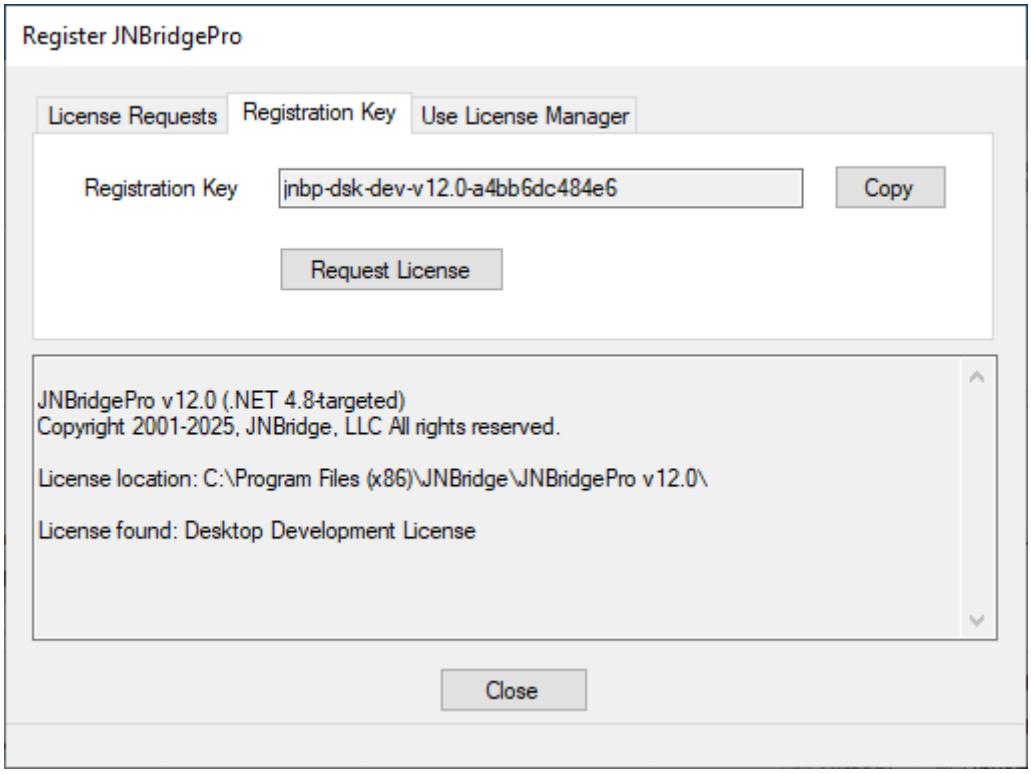


Figure 3. Registration Key

Click the "Request License" button and follow the instructions in the displayed page, or copy the displayed registration key into an email and mail the license request to registration@jnbridge.com, or visit https://jnbridge.com/support/license-key and follow the instructions there, supplying the registration key when requested. In response, you will be emailed a license file that you should deploy in the folder containing your application, or in the JNBridgePro installation folder.

License managers

Certain types of licenses, including floating licenses, as well as licenses that will run on terminal servers and virtual machines, require the use of a license manager. If you have purchased these licenses, JNBridge will supply you with a license manager, which you should install and configure according to instructions that are included with the license manager.

To use a license served up by a license manager, you will need to configure your application (or JNBridgePro proxy generation tool) to point to the license manager. See "Licensing and application configuration files," above, for a discussion of how to do this.

The registration tool provides some assistance in doing this. When the "Use License Manager" tab is selected (Figure 4), the user has the option of entering the hostname and port of the license manager (that is, the hostname of the machine on which the license manager resides, and the port on which it is listening). Once those values are entered, clicking on the "Load License" button will test whether these values are correctly configured, and, if they are, clicking on the "Export Config File" button will cause a fragment of XML containing the configuration information to be output to a file. The contents of the file can be incorporated into the application configuration file.

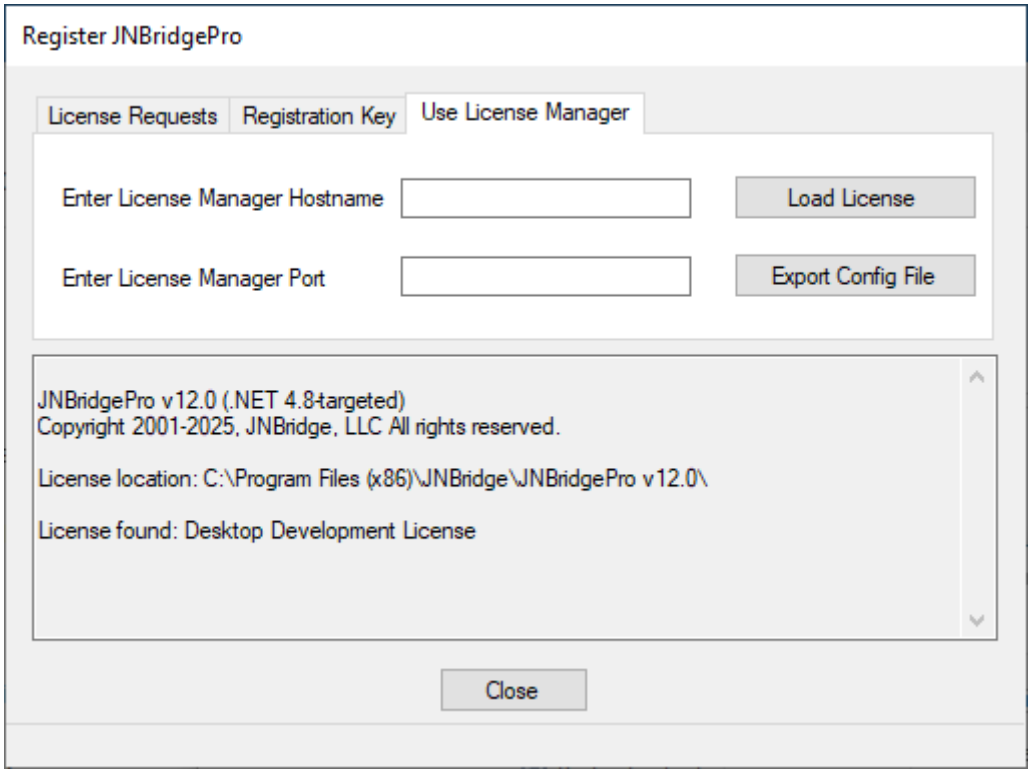


Figure 4. Configuring to use a license manager

It is also possible to place the license manager's host name and port number into a license file. The format of the file must be the following and the file must end in '.lic':

```
HOST [hostname] [port]
ISV jnbridge
```

Please note the following:

- Clicking the "Load License" button does not obtain a license for the application being licensed. It is still necessary to add information to the application configuration file that points to the license manager.
- If you have clicked on the "Load License" button, we recommend that you exit the registration tool before running your application.
- Use of the functionality in the "Use License Manager" tab is completely optional and is provided as a convenience. It is entirely possible to configure the application manually, without the assistance of this tool.

Licensing on Linux

In general, when running on Linux, the licensing mechanism works in the same way as with Windows and .NET. However, the following changes must be made. You must have root or sudo privileges to perform the following tasks.

- For 32-bit Linux, copy the file *jnbauth_x86.so* to */usr/lib*. For 64-bit Linux, copy the file *jnbauth_x64.so* to */usr/lib*. (Both *jnbauth_x86.so* and *jnbauth_x64.so* can be found in the *Linux* folder in the JNBridgePro installation.

The license file must either be in the directory where the .NET application running starts, or pointed to by the <*licenseLocation*> element added to the application's configuration file.

Licensing on .NET Core

In JNBridgePro for .NET Core, the development or deployment license file must either be in the same folder as the startup DLL, or its location must be indicated in the jnbridgeConfig.json file, in the licenseLocation:directory element. Alternatively, if you are using a license server, specify its location in jnbridgeConfig.json using the licenseLocation:host and licenseLocation:port elements.

The same JNBridgePro license file will work for both .NET Framework and .NET Core applications on any machine on which the license is valid.

The registration tool for .NET Core has a command-line interface only. To obtain the registration key for offline activation, please run

```
dotnet RegistrationTool.dll /regkey
```

and the registration key will be displayed.

Information on additional functionality in the registration tool can be obtained by running

```
dotnet RegistrationTool.dll /help
```

64-bit support

JNBridgePro supports 64-bit architectures and operating systems. We now provide a single installer with components and tools for both x86 (32-bit) and x64 (64-bit) processes. ***The single installer always runs as a 32-bit process, even on 64-bit machines, and will install by default with other 32-bit applications (that is, in the \Program Files (x86) folder). However, the installation will always contain both the 32-bit and 64-bit components, even on a 32-bit machine.***

Please note the following when using JNBridgePro on x64 systems:

- When using shared-memory communications on an x64 system, you must use an x64-targeted version of the Java Runtime Environment (JRE). When using tcp/binary, you may use either an x64-targeted version or an x86-targeted version.

- If you are deploying a 32-bit .NET application to an x64 system (that is, you are deploying a .NET application built to target the "x86" platform rather than targeted to '"x64" or "Any CPU"), if you are using shared memory, you should use a x86-targeted version of the JRE and the jnbsharedmem_x86.dll. If your application has been built as x64 and you are using shared memory, use jnbsharedmem_x64.dll. If your application has been built as "Any CPU" (so that it will run as 32-bit on a 32-bit platform, and as 64-bit on a 64-bit platform), use both jnbsharedmem_x86.dll and jnbsharedmem_x64.dll, and the correct one will always be used.

- If you are deploying an ASP.NET Web application to run on 64-bit ASP.NET and using shared-memory communications, you should use the x64-targeted assemblies. If you are using 32-bit ASP.NET and shared memory, you should use the x86-targeted assemblies.

- 64-bit support for shared memory is not available for .NET 1.x. If you have a .NET 1.x-targeted application and you want to deploy it to an x64 system, you can use tcp/binary, or you can deploy your 1.x-targeted application as a 32-bit 'x86' application (see above).

- When using the Visual Studio plug-in, and you use shared memory, you must use a 32-bit JRE (since Visual Studio is a 32-bit application). The standalone proxy tools are "Any CPU" and will run as 64-bit on 64-bit platforms, and as 32-bit on 32-bit platforms, so if you are using shared memory, choose the appropriate JRE.

- When creating Java proxies for Java-to-.NET projects, if you are attempting to proxy classes in a 32-bit (x86) DLL, we provide special 32-bit versions of the proxy generation tools, but we only recommend using them for that specific purpose.

.NET Core and .NET 8 support

The following features of "traditional" JNBridgePro are not available in JNBridgePro for .NET Core:

- Failover.
- Embedding of Java UI elements in .NET Windows Forms and WPF applications
- Embedding of .NET Windows Forms and WPF elements in Java AWT, Swing, or SWT applications.

System Requirements

The .NET side of applications using JNBridgePro must run on .NET Core 3.0 or 3.1. The Java side must run on Java 5 through Java 17.

The .NET side can run on any Windows or Linux platform on which .NET Core 3.0 is supported (except for shared memory on ARM32, ARM64, and Alpine Linux). (MacOS X will be supported in a future version.) For a full list of supported platforms, see https://github.com/dotnet/core/blob/master/release-notes/3.0/3.0-supported-os.md or similar Microsoft documentation. Depending on the platform (as indicated in the abovementioned documentation), 32-bit, 64-bit, or both, may be supported.

The Java side can run on any platform that that supports Java 5 or later. Please see Oracle documentation for further information.

JNBridgePro and Security

Use of TCP/binary communications in JNBridgePro potentially exposes certain security vulnerabilities. In .NET-to-Java projects, for example, the Java side awaits requests from clients to execute Java APIs and return results. Ideally, these clients are .NET applications under your control, accessing the .NET side through proxies that you have created. However, it is possible for a malicious application to act as a client, accessing the Java side through JNBridgePro's TCP/binary protocol. Proxies are not required (nor is .NET), and even APIs that have not been proxied can be accessed; in particular, the Runtime.exec() API can be called, leading to the ability to execute arbitrary code on the Java-side machine. A similar vulnerability exists when using TCP/binary communications in Java-to-.NET projects to access the .NET-side machine.

In order to avoid these vulnerabilities, JNBridgePro offers a number of security features, some of which were introduced starting with JNBridgePro version 10.1. IP and class whitelisting are turned on by default; shared memory and SSL must be explicitly chosen. While you may turn any of these features off, you should be aware of the risks involved, and should be confident that you are otherwise mitigating these vulnerabilities.

The following is a summary of JNBridgePro's security features:

- ***Shared memory:*** The most secure way to use JNBridgePro is through shared memory, in which the .NET and Java sides run in the same process, and in which communications between the two sides is through shared data structures rather than through sockets. Because the remote Java or .NET side is not exposed to the network through sockets, the TCP/binary vulnerability does not exist. If use of shared memory is feasible for your scenario (i.e., you have exactly one .NET side and one Java side in your application, and they reside on the same machine), you should always use it. In addition to better security, shared memory is also much faster than TCP/binary communications, and (unlike with TCP/binary) shared memory automatically starts the remote Java or .NET side when the application starts, and shuts it down when the application ends.
- ***SSL communications:*** If TCP/binary communications must be used, consider using SSL (Secure Sockets Layer) for the connection between the Java and .NET sides. JNBridgePro's implementation of SSL in TCP/binary requires both client and server authentication. This guarantees that a JNBridgePro client (the Java or .NET side making the cross-platform calls) is communicating with the intended server (the .NET or Java side responding to the cross-platform calls) rather than some malicious server acting in its place, and it also guarantees that only authorized JNBridgePro clients that possess the appropriate credentials can access the server. It also guarantees an encrypted connection between the .NET and Java sides. SSL is turned off by default, since is can be more complex to configure than the other features, but you should consider turning it on when using TCP/binary communications. Once it is turned on, it *must* be configured before it will work. For instructions on configuring SSL, see the sections "Secure communications using SSL" in the .NET-to-Java and Java-to-.NET chapters in the *Users' Guide*.
- ***Class whitelisting:*** An effective way to mitigate the TCP/binary vulnerability is to limit the APIs that can be accessed from a remote client. JNBridgePro allows you to specify a whitelist of classes whose APIs can be accessed remotely. Accesses to any classes not in the whitelist will be rejected and an exception will be thrown. A judiciously chosen whitelist containing only those classes that your clients will be accessing will substantially mitigate the risks of the TCP/binary vulnerability, but it is still possible for a malicious client to access whitelisted classes. If that poses a risk because those classes implement sensitive APIs, then you should use class whitelisting together with SSL. Class whitelisting is turned on by default, and a class whitelist file should be supplied if you want to whitelist more than the small list of classes that are always whitelisted. Class whitelisting can be turned off, but this must be done explicitly. For instructions on class whitelisting and how to configure it, see the sections "Class whitelisting" in the .NET-to-Java and Java-to-.NET chapters in the *Users' Guide*.
- ***IP whitelisting:*** One rudimentary way to mitigate the TCP/binary vulnerability is to limit the hosts from which a Java-side or .NET-side server will accept cross-platform requests. JNBridgePro allows you to specify a whitelist of those IP addresses from which requests will be accepted; requests from all other hosts will be rejected. The whitelist can include wildcards, so that subnets and other ranges can be whitelisted. IP whitelisting does not prevent access by malicious clients residing on the whitelisted hosts, so it should generally be used in conjunction with other security features like SSL and class whitelisting. IP whitelisting is turned on by default, with requests only accepted from clients on the same machine as the server. It can be explicitly turned off by supplying a complete wildcard whitelist, but you should only consider doing this if you are using SSL and/or class whitelisting, or have extreme confidence in your computing environment, including (but not restricted to) a properly configured network firewall. For instructions on IP whitelisting, see the sections "Whitelisting .NET clients" and "Whitelisting Java clients" in the *Users' Guide*.

Note that the JNBridgePro proxy generation tools use TCP/binary communications along with same-host IP whitelisting and a highly-restricted class whitelist, so use of TCP/binary in this context should be considered safe.

Using JNBridgePro for .NET-to-Java calls

This section describes how to use JNBridgePro when using it in the *.NET-to-Java* direction. Both *proxy generation* and *proxy use* scenarios are described. For information on using JNBridgePro in the *Java-to-.NET* direction see "Using JNBridgePro for Java-to-.NET calls", and for information on *bi-directional* use see "Use of JNBridgePro for bi-directional interoperability."

Generating proxies with JNBridgePro

Each Java class that is to be accessible to .NET classes must have a corresponding .NET proxy class that manages communications with the Java class. .NET classes interact with the proxy class as though it were the underlying Java class, and the actions are transparently communicated to the underlying Java class. Creation of these proxy classes is typically done during application development, and is accomplished through use of the JNBProxy tool included in JNBridgePro.

JNBProxy will, given one or more Java class names, generate .NET proxies for those classes, and optionally all the supporting classes, including parameter and return value types, thrown exceptions, interfaces, and super classes, resulting in complete class closure. One can also restrict JNBProxy to generate proxies for only specified classes, and not for the entire set of supporting classes.

Note: You cannot generate a proxy for an obfuscated Java class unless you know the obfuscated name of the class. Similarly, you cannot access an obfuscated member (a field or method) of a class unless you know the obfuscated name of the member.

Use of JNBProxy in the .NET-to-Java direction results in creation of a .NET assembly (a DLL file) containing the implementation of the .NET proxy classes corresponding to the Java classes. Once this assembly has been generated, it can be used to access the corresponding Java classes as described in the section *Using proxies with JNBridgePro.*

JNBProxy comes in three forms: a standalone GUI-based application, a Visual Studio plug-in (for .NET-to-Java projects; for Java-to-.NET projects, there is an Eclipse plug-in), and a command-line-based standalone application. These are described below.

JNBProxy (standalone GUI-based application)

The **JNBProxy** Windows (GUI) application (`jnbproxygui.exe`) can be used to generate .NET proxies for Java classes.

Initial invocation

The first time the JNBProxy GUI is invoked after installation, its Java options must be configured. JNBProxy displays a copy of the Java Options window, as in Figure 5 below. (Note: you may reconfigure these values later via the **Project→Java Options…** menu item, which brings up the same window.)

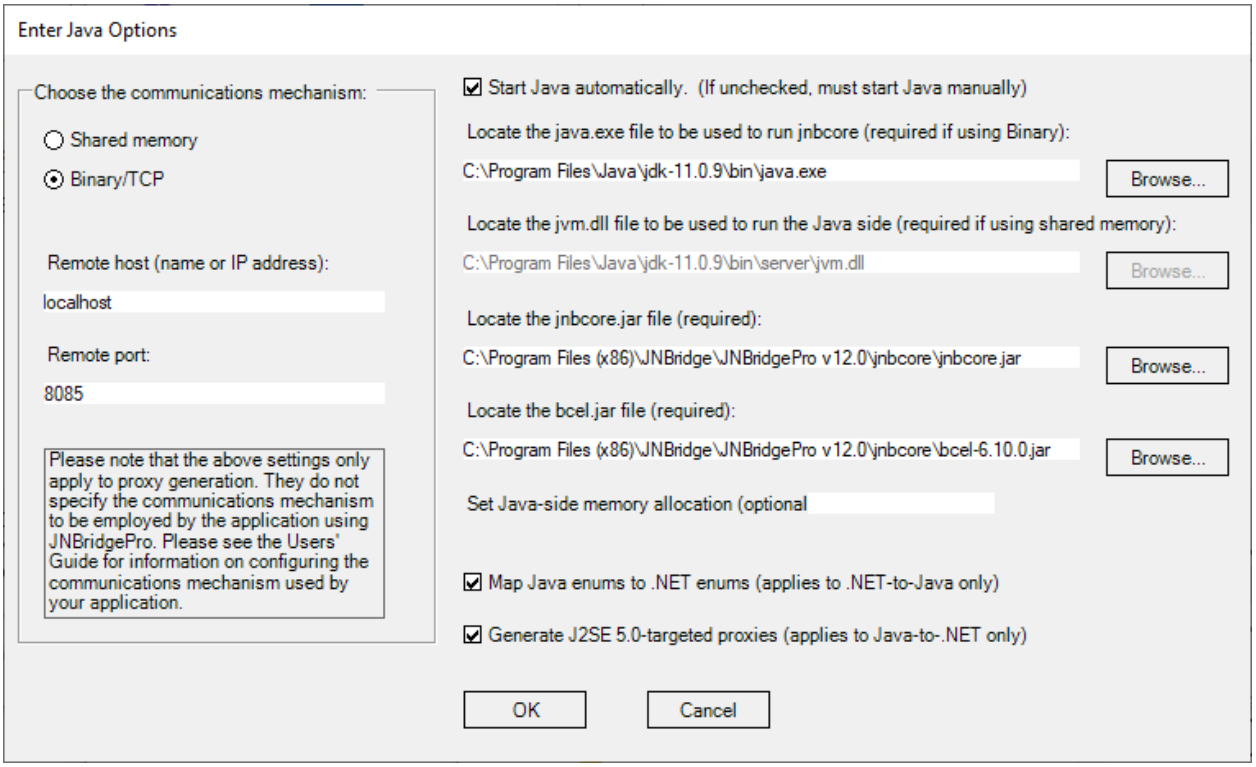


Figure 5: Java Options window

JNBProxy will attempt to fill in these options itself. Typically, there is no need to do anything further; the supplied choices will be sufficient. However, if JNBProxy cannot find a necessary file, or if unusual conditions apply, you may need to explicitly specify one or more of the options.

Two communications mechanisms are available for managing the communications between the .NET and Java sides: shared memory, and binary/TCP. If you select binary/TCP, you must specify the host on which the Java side resides, and the port through which it will listen to requests from the .NET side. If you select shared memory, you must locate the file jvm.dll that implements the Java Virtual Machine that will be run in-process with the .NET-side code. Typically, jvm.dll is included as part of the JDK (Java Development Kit) or JRE (Java Runtime Environment) on which you will be running the Java side.

The default settings (binary/TCP, localhost, and port 8085) are typically sufficient and usually need not be changed.

Note: While shared memory is faster than binary/TCP, it has not been chosen as the default setting for proxy generation due to certain limitations in the Java Native Interface (JNI), which require that JNBProxy be exited and restarted whenever JNBProxy's classpath is changed.

If Java is to be started automatically, you must supply the location of the Java runtime (`java.exe`), along with the locations of the `jnbcore.jar`, `jnbcore.properties`, and the BCEL jar files. (Note that the BCEL jar file is only used in the *Java-to-.NET* direction, but its location must always be specified.)

If you wish to start Java manually, uncheck the "Start Java automatically" box.

Note: if the Java side is located on another machine, JNBProxy cannot start it automatically; it must be started manually.

When JNBProxy is launched, the application displays a *launch form* (Figure 6). Depending on which radio button is selected, the user can create a new .NET-to-Java project, a new Java-to-.NET project, a recently-used project, or some other existing project file. After the project is selected, the main form of JNBProxy will appear (Figure 7).

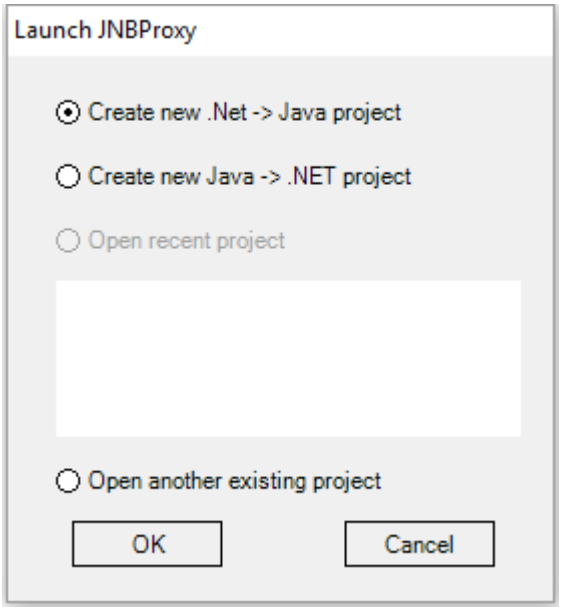


Figure 6. JNBProxy launch form

GUI layout

Figure 7 shows the GUI version of the JNBProxy tool immediately after it is launched. Within the tool's window are four panes: the *environment pane*, the *exposed proxies pane*, the *signature pane*, and the *output pane*.

The environment pane displays the Java classes of which jnbproxy is aware, and for which proxies can be generated. The exposed proxy pane shows those Java classes for which proxies will be generated. The signature pane shows information on the methods and fields offered by a Java class, as well as other information on the class. The output pane displays diagnostic and informative messages generated during the operation of the tool. You may drag the splitter controls to resize the panes.

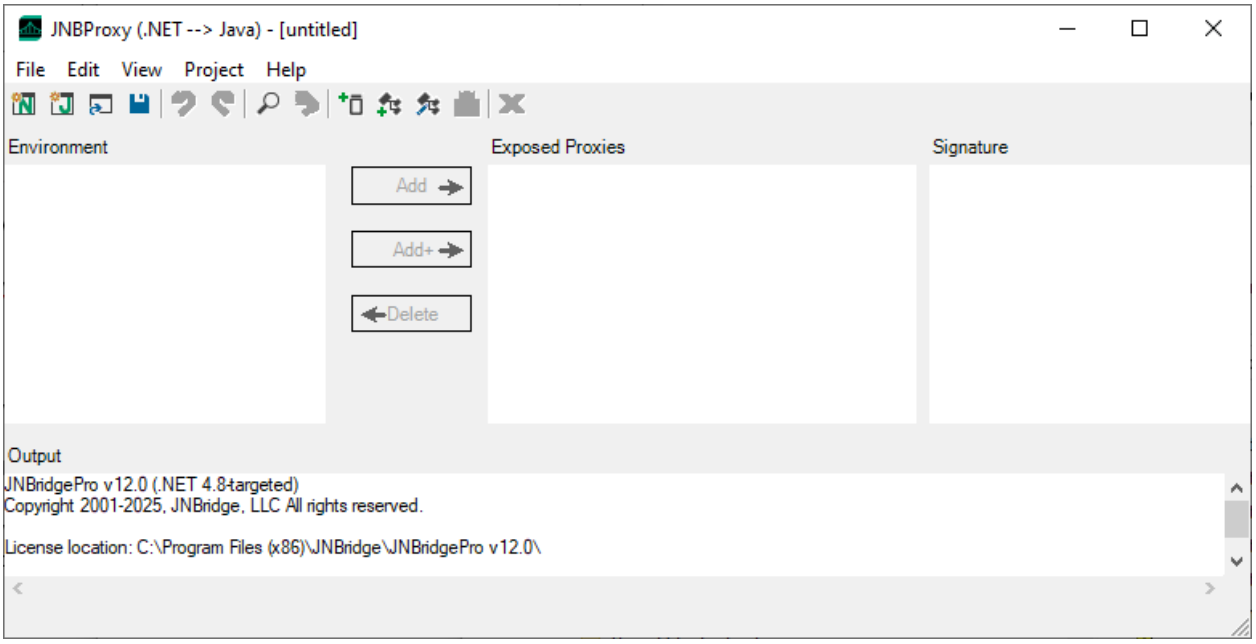


Figure 7. JNBProxy proxy generation tool (GUI version)

The title bar of JNBProxy contains an indication of whether the current project is Java-to-.NET or .NET-to-Java. In Figure 7, we see that the current project is .NET-to-Java.

Process for generating proxies

Generating proxies takes three steps:

- Add classes to the environment for which you might want to generate proxies.
- Select classes in the environment for which proxies are to be generated and add them to the exposed proxies list.
- Build the proxies into a .NET assembly.

Adding classes to the environment

The first step in generating proxies is to load candidate classes into the environment. Think of the environment as a palette from which one can choose the proxies that are actually generated. Classes can be loaded into the environment in two ways: from a jar file, and as specific classes found along the Java classpath.

To load classes from a jar file, click on the menu item **Project→Add classes from JAR file…**. A dialog box will appear allowing the user to locate one or more jar files whose classes will be loaded. Opening a jar file will cause all the classes inside of it to be added to the environment pane. The progress of the operation is shown in the output pane. The operation can be terminated before completion by clicking on the Stop button located on the GUI's tool bar.

Note: any jar file from which classes are to be loaded must be in the Java classpath. To edit the classpath, see "Setting or modifying the Java classpath," below.

Note: The results of operations terminated prematurely by clicking the stop button may be inconsistent and should not be used. These partial results can be removed by performing an Undo operation.

To load specific classes from the classpath, click on the menu item **Project→Add classes from classpath…**. When this item is selected, the Add Classes dialog box is displayed (Figure 8).

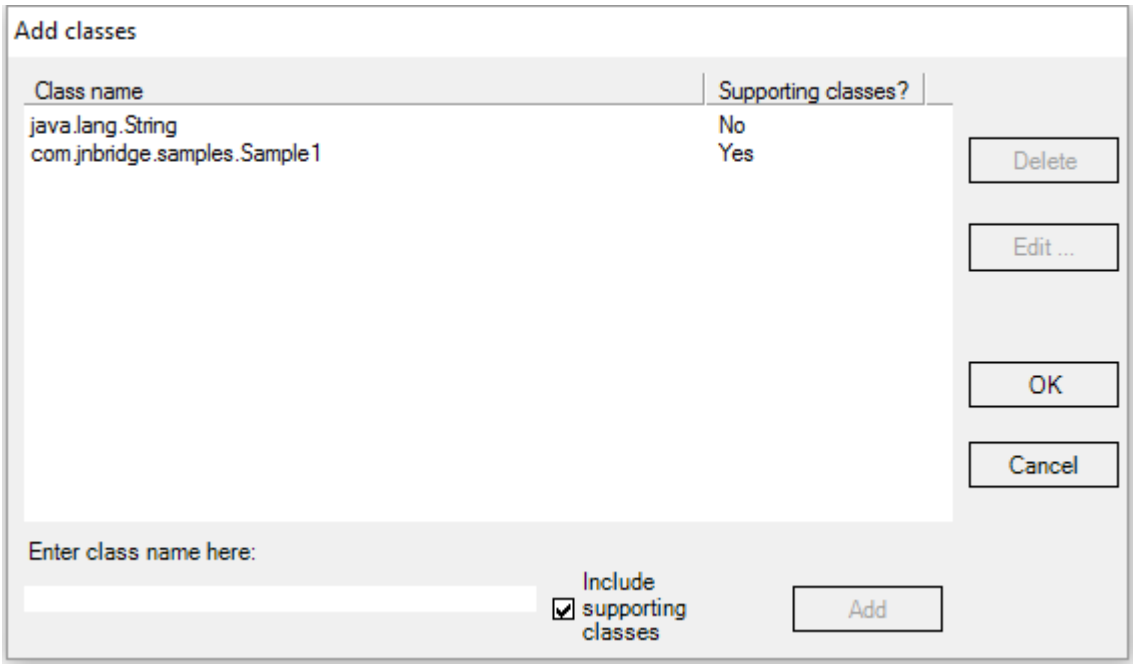


Figure 8. Add classes dialog box

To specify a class to be added, type the fully-qualified class name into the text box at the bottom of the dialog box. (Note that, as you type, the interface will provide class name suggestions based on what is available and what you have typed so far.) Leave checked the "Include supporting classes" check box to indicate that all classes supporting the specified class should be automatically added. (See *Supporting classes*, below, for more information on supporting classes.) Then, click the Add button to add the class to the list of classes to be loaded into the environment. You may delete a class from this list by selecting the class in the list and clicking on the Delete button. You may edit the information for the class by selecting the class and clicking on the Edit button or by double-clicking on the class. When you are done specifying classes, click the OK button to add the classes to the environment. This process may take a few minutes, especially if any of the classes in the list must also have their supporting classes loaded. The operation can be terminated before completion by clicking on the Stop button located on the GUI's tool bar.

Also note that, in addition to specifying the fully-qualified name of a class to be added, you can also use a trailing '*' as a wildcard, to indicate that all classes in the classpath that match begin with the string that precedes the asterisk should be added. For example, supplying 'java.util.*' indicates that all the classes in the java.util.* package should be added.

Note: If you are doing Java-.NET co-development, and a Java class is changed after the class has been loaded into the environment (e.g., a method or field has been added or removed, or the signature changed), the class can be updated in the environment simply by loading it again.

Note: Any class or supporting class that is added must be in the classpath or a member of one of the standard java.* packages.

Note: Proxies for java.lang.Object and java.lang.Class are always generated and added to the environment even if they are not explicitly requested or implicitly requested by checking "Include supporting classes."

Note: The results of operations terminated prematurely by clicking the stop button may be inconsistent and should not be used. These partial results can be removed by performing an Undo operation.

Selecting proxies to be generated

Once classes have been loaded into the environment, they are displayed in the environment pane, which is a tree view listing all the classes loaded into the environment grouped by package. Next to each item in the tree view is an icon indicating whether the item is a package, an interface, and exception, or some other class. (See Figure 9.)

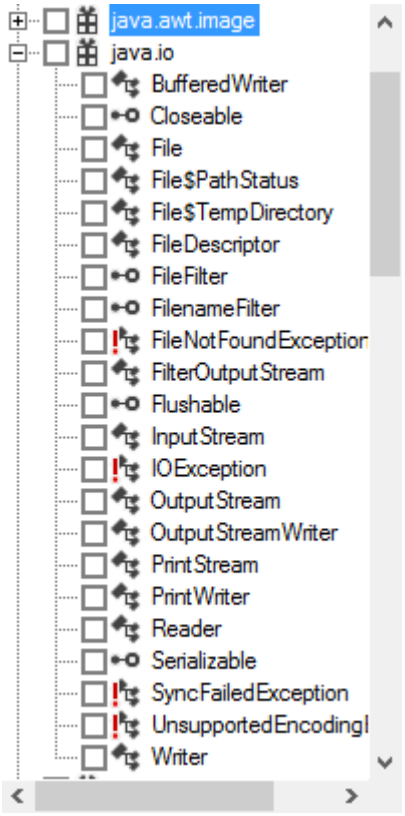


Figure 9. Environment pane

To select a class for proxy generation, check the class's entry in the environment pane by clicking on the check box next to the name. To select all the classes in a package, check the package's entry by clicking on its check box. Once a set of classes has been checked, add them to the set of exposed proxies by clicking on the **Add** button.

Alternatively, you may click on the **Add+** button to add the checked classes and all supporting classes. For a discussion on supporting classes, see the *Supporting Classes* section later in this guide. The checked classes and all supporting classes (if **Add+** was clicked) will appear in the exposed proxies pane, which is a tree view similar to the environment pane.

Note: use of* Add+ *may take a few minutes for the operation to complete. The operation can be terminated before completion by clicking on the Stop button located on the GUI's tool bar. The results of operations terminated prematurely by clicking the stop button may be inconsistent and should not be used. These partial results can be removed by performing an Undo operation.

To selectively remove classes from the exposed proxies pane so that proxies are not generated, check the entries in the exposed proxies pane for all classes or packages to be removed (by clicking on the check boxes next to the class or package names), then click on the **Delete** button. The checked items will be removed from the exposed proxies pane.

As a convenience, the Edit menu contains items to check or uncheck all the items in the environment or exposed proxy panes.

Add, **Add+**, and **Delete** operations may be repeatedly performed until the exact set of proxies to be generated appears in the exposed proxies pane.

The most recent **Add**, **Add+**, and **Delete** operations may be undone and then redone using the **Edit→Undo** and **Edit→Redo** menu items.

Note: Proxies for java.lang.Object and java.lang.Class will always be added to the exposed proxies pane when the Add button is clicked, even if they have not been checked in the environment pane. Also, java.lang.Object and java.lang.Class cannot be checked in the exposed proxies pane, and therefore cannot be deleted from that pane. The reason for this behavior is to ensure that proxies for Object and Class are always generated.

Designating proxies as reference or value

Before generating proxies for the classes listed in the exposed proxies pane, the user can designate which proxies should be reference and which should be value. (See the section "Reference and value proxies," below, for more information on the distinction between reference and value proxies.) The default proxy type is reference; if the user wants all proxies to be reference, nothing additional need be done.

To set a proxy's type (i.e., to reference or any of the three styles of value), position the cursor over the proxy class to be changed (in the exposed proxies pane), and right-click on the class. A pop-up menu will appear. Choose the desired proxy type. After the proxy type is selected, the proxy class will be color coded to indicate its type (black for reference, blue for value (public/protected fields style), red for by-value (JavaBean style), or green for by-value (mapped).

To set the types of multiple proxy classes at the same time, click on the **Project→Pass by Reference / Value…** menu item. The Pass by Reference / Value dialog box is displayed (Figure 10). All proxy classes in the exposed proxies pane are listed in this dialog box. To change the types of one or more classes, select those classes (left-click on a class to select it, and shift-left-click or ctrl-left-click to do multi-selects), then click on the Reference, Value (Public/protected fields), Value (JavaBeans) or Value (Mapped) button to associate the selected classes to the desired type. When done, click on the OK button to actually set the classes to the chosen types and dismiss the dialog box, click on the Apply button to set the classes without dismissing the dialog box, or click on the Cancel button to discard all changes.

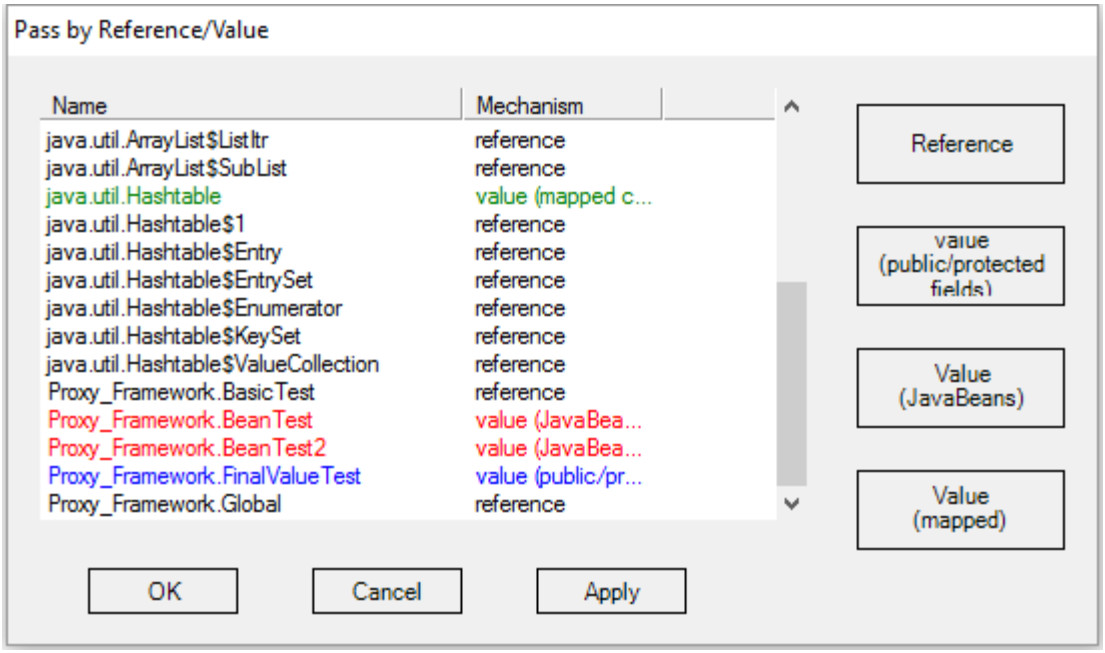


Figure 10. Pass by Reference/Value dialog box

Designating proxies as transaction-enabled (formerly "thread-true")

Before generating proxies for the classes listed in the exposed proxies pane, the user can designate which proxies should be *transaction-enabled*. (See the section "Transaction-enabled classes and support for transactions," below, for more information on transaction-enabled classes.) The default proxy type is not transaction-enabled; if the user wants all proxies to be non-transaction-enabled, nothing additional need be done.

To set a proxy as transaction-enabled, position the cursor over the proxy class to be changed (in the exposed proxies pane), and right-click on the class. A pop-up menu will appear. Select the "transaction enabled" menu item so that it is checked. To remove the transaction-enabled property from a proxy class, perform the same operation to uncheck the transaction-enabled menu item. It is currently not possible to make multiple classes transaction-enabled at the same time.

Users should use the transaction-enabled property sparingly, as it incurs a performance penalty. See the section "Transaction-enabled classes and support for transactions" for more information.

Generating the proxies

Once classes have been selected for proxy generation and have been moved into the exposed proxies pane, a .NET assembly with the proxies can be generated by clicking on the **Project→Build…** menu item. A dialog box will be displayed that allows the user to specify the location and name of the .NET assembly that will contain the generated proxies. If one or more proxy classes are value, a consistency check will be performed before the proxies are generated (see "Reference and value proxies," below), and the results will be reported to the user.

Note: The Build operation may take a few minutes for the operation to complete. The operation can be terminated before completion by clicking on the Stop button located on the GUI's tool bar. The results of operations terminated prematurely by clicking the stop button may be inconsistent and should not be used.

Saving and restoring projects

Work done during a JNBProxy project session can be saved in a JNBProxy project file. A project file contains the current Java classpath and the current contents of both the environment and exposed proxies panes. A project file is loaded by clicking on the menu item **File→Open Project…**, which opens a dialog that allows the user to identify the project file to be opened. If the current project has been modified, the user is offered the opportunity to save it.

Alternatively, the **File→Recent Projects** menu item allows access to the four most recently used projects.

A project is saved by clicking on the menu item **File→Save Project** or **File→Save Project As…**. The latter option lets the user specify the name and location of the project file, while the former option saves the session under its current name if it has one; otherwise it asks the user to specify the name and location of the new project file.

A new project is created by clicking on the menu item **File→New .NET->Java Project…** or **File→New Java->.NET Project…**.depending on the desired direction of the new project. If the current project has been modified, the user is offered the opportunity to save it.

Exporting a class list

Selecting the **File→Export classlist…** menu item will cause a text file to be generated that lists the classes in the Exposed Proxies pane, plus information on whether the classes are by-reference or by-value, and whether the classes are transaction-enabled. The exported class list is in the correct format to be used as input with the '/f' option of the command-line version of JNBProxy. (See the section "Using JNBProxy (command-line version)").

Generating a command-line script

In certain cases, it will be useful or necessary to incorporate proxy generation into an automated build process. The command-line version of the proxy generator (see below) is intended for these situations. To simplify the creation of command-line scripts, JNBProxy can be used to automatically generate a script for the current project. Select the **JNBridgePro→Generate command-line script…** menu item to generate a .bat file that can be used to invoke the command-line proxy generator. When generating a command-line script, the user will be prompted to supply the name of the .bat file. At the same time, a class list will be generated (see "Exporting a class list," above). If the .bat file is named *myFile.bat*, the class list will be named *myFile_classlist.txt*, unless a file of that name already exists, in which case the user will be asked whether it should be overwritten, or whether the file should have a different name.

Note that the .bat file can, and in many cases should, be edited, particularly if it will be run in a different location from that to which it was written. For information on the command-line options and how they can be edited, please see the section "Using JNBProxy (command-line version)," below.

Examining class signatures

JNBProxy displays information about a Java class so that a user can determine whether it contains properties of interest to the user. If a class item in either the environment or exposed proxies pane is selected, its information is displayed in the signatures pane. The information includes the class's name, superclass, attributes, interfaces, fields, and method signatures (Figure 11).

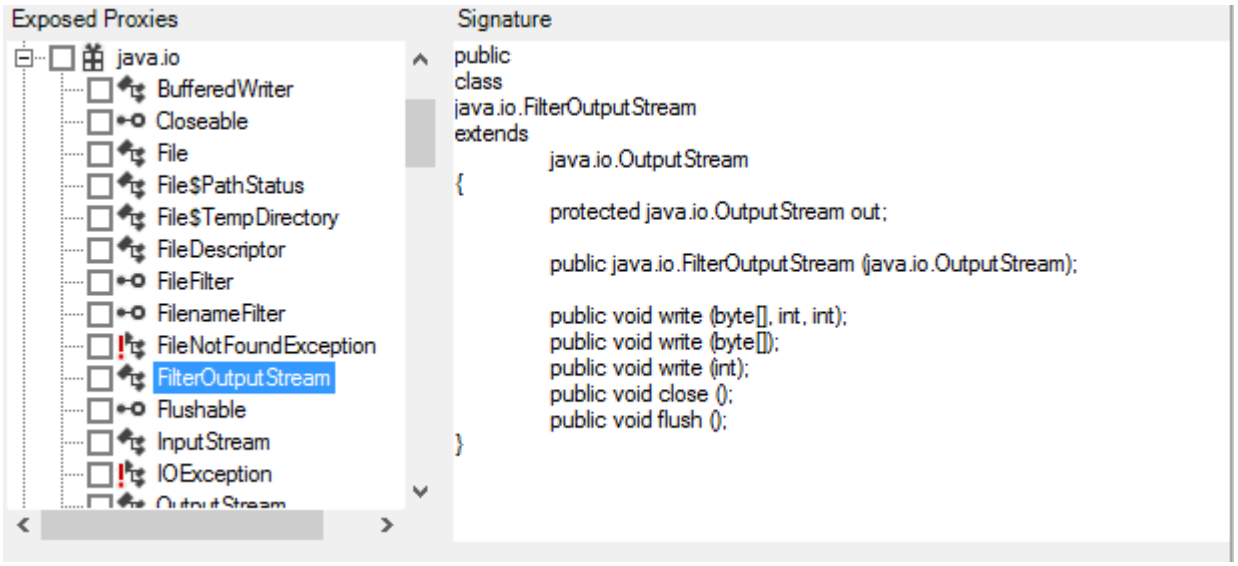


Figure 11. Selected item in exposed proxy pane and corresponding information in signature pane

Searching for classes in the environment or exposed proxy panes

To find a class among a large number of classes in the environment or exposed proxy panes, use the Find facility available by clicking on the **Edit→Find** menu item, which displays a Find window (Figure 12). The Find window offers a variety of options, including the ability to search the environment or exposed proxy panes, to search down or up, to look for exact whole-word matches, and to perform case-dependent matches. To repeat a search, the user should use the **Edit→Find Next (F3)** menu item.

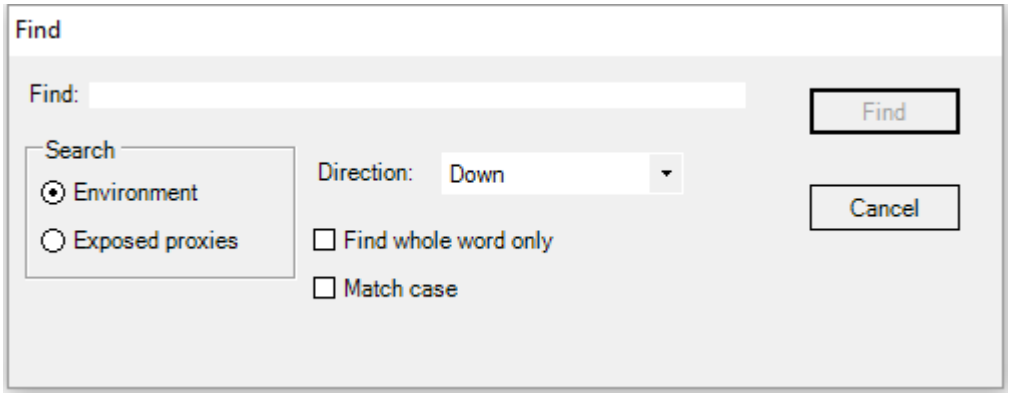


Figure 12. Find window.

Modifying tree view properties

To change the ways that the environment and exposed proxy panes are displayed, click on the **View→Tree Options…** menu item. When this option is selected, a dialog box is displayed that allows the user, for either the environment or exposed proxy pane, to show or hide the icons in the pane, and to completely expand or completely collapse the tree view in the pane (Figure 13).

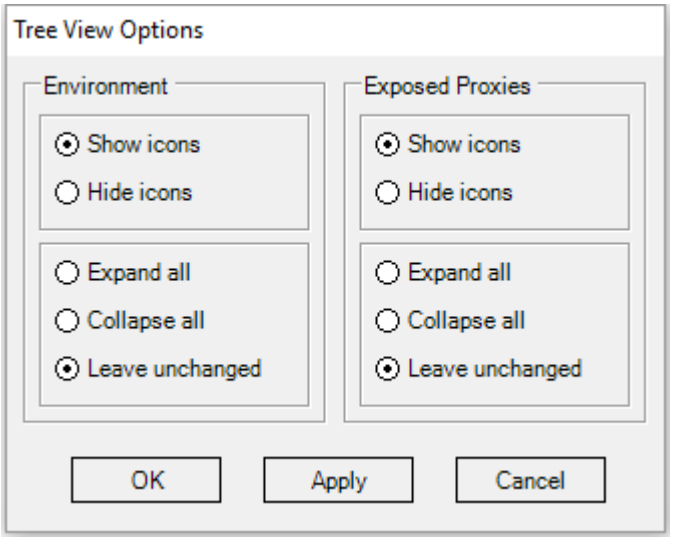


Figure 13. Tree options window

Refreshing the display

To refresh the contents of the four panes (environment, exposed proxy, signature, and output) in the JNBProxy window, select the **View→Refresh (F5)** menu item.

Setting Java startup options

The user may choose for JNBProxy to automatically start the Java side, or the user may start it manually. (See *Starting Java manually* for more information.) To control whether or not Java is started automatically, and to specify the location of various executables and libraries, select the **Project→Java Options…** menu item (Figure 14).

Two communications mechanisms are available for managing the communications between the .NET and Java sides: shared memory, binary/TCP. If you select binary/TCP, you must specify the host on which the Java side resides, and the port through which it will listen to requests from the .NET side. If you select shared memory, you must locate the file jvm.dll that implements the Java Virtual Machine that will be run in-process with the .NET-side code. Typically, jvm.dll is included as part of the JDK (Java Development Kit) or JRE (Java Runtime Environment) on which you will be running the Java side.

The default settings (binary/TCP, localhost, and port 8085) are typically sufficient and usually need not be changed. **Note that these settings only apply to proxy generation, not to the way the Java and .NET sides will communicate in the application under development. To configure the communication in the application you are developing, see the section "Using Proxies with JNBridgePro."**

Note: While shared memory is faster than binary/TCP, it has not been chosen as the default setting due to certain limitations in the Java Native Interface (JNI), which require that JNBProxy be exited and restarted whenever JNBProxy's classpath is changed.

To cause Java to be started automatically by JNBProxy, check the check box at the top of the Java options window. Leave the box unchecked if you wish to start Java manually.

If Java is to be started automatically, you must supply the location of the Java runtime (`java.exe`), along with the locations of the `jnbcore.jar` and `jnbcore.properties` and `bcel-6.n.m.jar` files. (`bcel-6.n.m.jar` is only used in the Java-to-.NET direction, but its location must always be specified.)

If you wish to start Java manually, uncheck the "Start Java automatically" box.

Note: if the Java side is located on another machine, JNBProxy cannot start it automatically; it must be started manually.

If the user's Windows account contains privileges allowing it to write new settings to the registry, the Java Options settings will be saved from one invocation of JNBProxy to the next; if the account does not have such privileges, the settings will not be saved.

Adjusting the Java-side memory allocation during proxy generation

When generating proxies for a very large number of proxies (for example, all the classes in a very large jar file), it is possible that the Java side may run out of memory, and proxy generation will fail. When this happens, one can avoid the problem by specifying a larger Java-side memory allocation to be used during proxy generation.

To adjust the Java-side memory allocation, bring up the Java Options dialog box by selecting the **Project→Java Options…** menu item. The Java Options window (Figure 14) will be displayed. One can specify a new Java-side memory allocation by entering it in the box labeled "Set Java-side memory allocation." The value entered there is the new Java-side memory allocation in bytes, and may be either a number that is a multiple of 1024, a number followed by "m" or "M" (denoting megabytes), or a number followed by "k" or "K" (denoting kilobytes). If the entry is left blank, the default memory allocation for the current Java runtime environment will be used.

Note: The Java-side memory allocation setting only applies to proxy generation. It does not affect subsequent use of the proxies. To specify a Java-side memory allocation to be used during the running application, you need to supply a –Xmx option to the Java side.

Note: If the "Start Java automatically" checkbox is unchecked, the Java-side memory allocation option will be ignored.

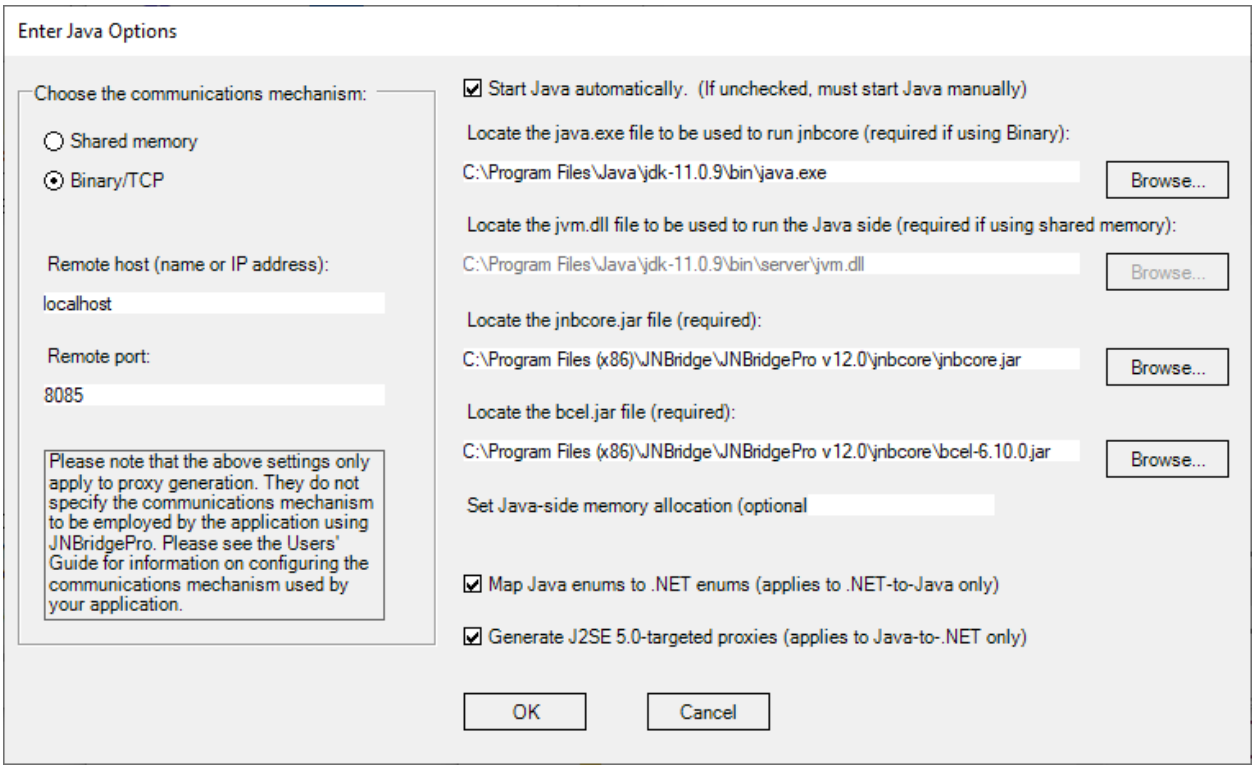


Figure 14. Java Options window.

Proxying Java enums

Originally, Java enum types were proxied in the same way as any other Java type. The proxied enums would inherit from the proxy of java.lang.Enum, and all of the enum's associated methods could be accessed through the proxies. Later versions of JNBridgePro mapped Java enums to .NET enums during proxy generation. While this made accessing the enumerated values much more efficient, it resulted in loss of access to the methods in the underlying Java enums, and it made it impossible to proxy a Java class that was a subclass of an enum (something that is legal in Java), because a subclass of a .NET enum is not permitted. It also made it difficult for some users to upgrade to newer versions of JNBridgePro.

With the current version of JNBridgePro, the user can control the way in which Java enums are proxied. A checkbox in the Java Options dialog box (Figure 14) labeled "Map Java enums to .NET enums" controls this behavior. The default is for the box to be checked, in which case the Java enums are mapped to .NET enums. To simply proxy Java enums in the same way as other classes, uncheck this box before generating the proxies.

This check box only applies to .NET-to-Java projects. It is ignored in Java-to-.NET projects.

Setting or modifying the Java classpath

If Java is started automatically, the Java classpath should be set. Do this by selecting the **Project→Edit Classpath...** menu item. This will call up the Edit Classpath window (Figure 15).

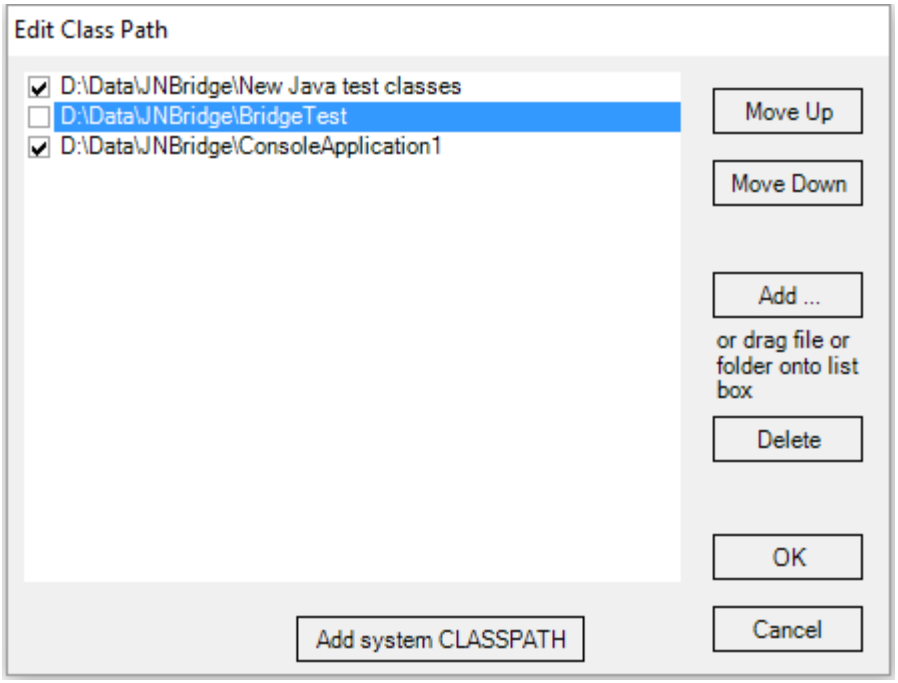


Figure 15. Edit Classpath window.

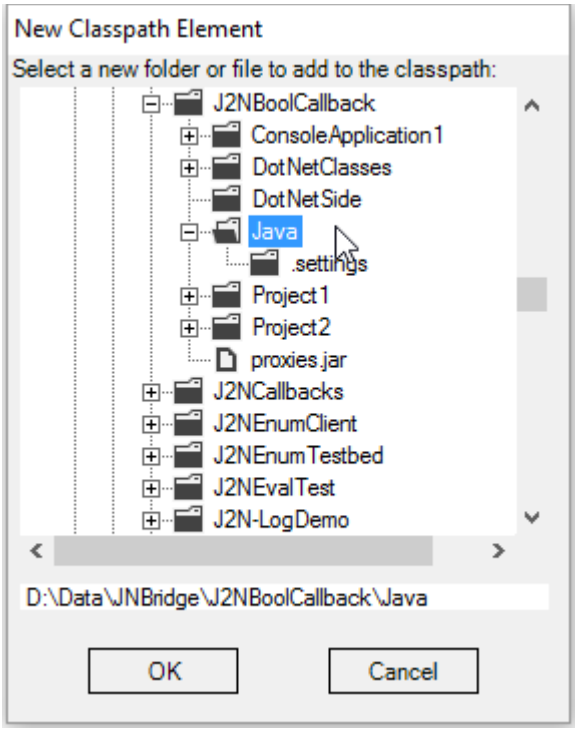


Figure 16. New Classpath Element window.

The classpath can be rearranged by selecting a classpath element and clicking on the **Move Up** or **Move Down** buttons, or by selecting an element and clicking on the **Delete** button. The files and folders in the system CLASSPATH environment variable can be added to the classpath list by clicking on the **Add system CLASSPATH** button. Note that only checked elements will be added to the Java classpath when the dialog is dismissed.

To add a folder or jar file to the classpath, simply drag and drop one or more files or folders onto the Edit Classpath form. If you drag a folder, you will be given the choice of simply adding the folder to the classpath (useful if you are developing Java using .class files), or of adding all the jar files contained in the folder to the classpath. You are also given the option of telling the proxy generation tool to make the same choice for all subsequent adds of folders. (If you select this option, you can always change that decision by selecting **Project→JNBProxy Options…** and selecting "Always ask" as the "Folder drag-and-drop" action.

You can also add a folder or jar file by clicking on the **Add…** button. This causes a New Classpath Element window to be displayed. In this window, the user can navigate to the desired folders or jar files, or can enter a file path directly (Figure 16). The New Classpath Element window supports multiple selection - multiple folders and/or jar files may be selected by ctrl-clicking, while a range of folders and/or jar files may be selected by shift-clicking. Clicking on the **OK** button will cause the indicated folders or files to be added to the Edit Classpath window.

Note: You can also type a directory or file path directly into the New Classpath Element dialog box. This path can be an absolute path, or a UNC path to a shared network folder (e.g., \\MachineName\FolderName\FileName.jar).

If the classpath is modified while the Java side is running, and automatic Java startup is set, the Java side is automatically halted and restarted.

Note: if the Java side is started up manually, changing the JNBProxy classpath will have no effect.

Note: If Java is started up automatically, terminating JNBProxy with an external kill may leave a stray Java process running that you will have to terminate as well.

Note: If shared-memory communication between the .NET and Java sides is used, it will be necessary to exit and restart JNBProxy whenever the classpath is changed. This is due to limitations in the Java Native Interface (JNI), which is used by the shared-memory channel.

Strong naming proxy assemblies

.NET supports the notion of *strong naming* of assemblies, which includes the assignment of version numbers as well as digitally signing the assemblies. For more details on strong naming of assemblies, see the .NET Framework documentation.

To strong name a proxy assembly, the strong naming options must be set before the proxy assembly is built. To set strong naming options, select the **Project→Strong Naming Options…** menu item. The Strong Naming dialog box will be displayed (Figure 17).

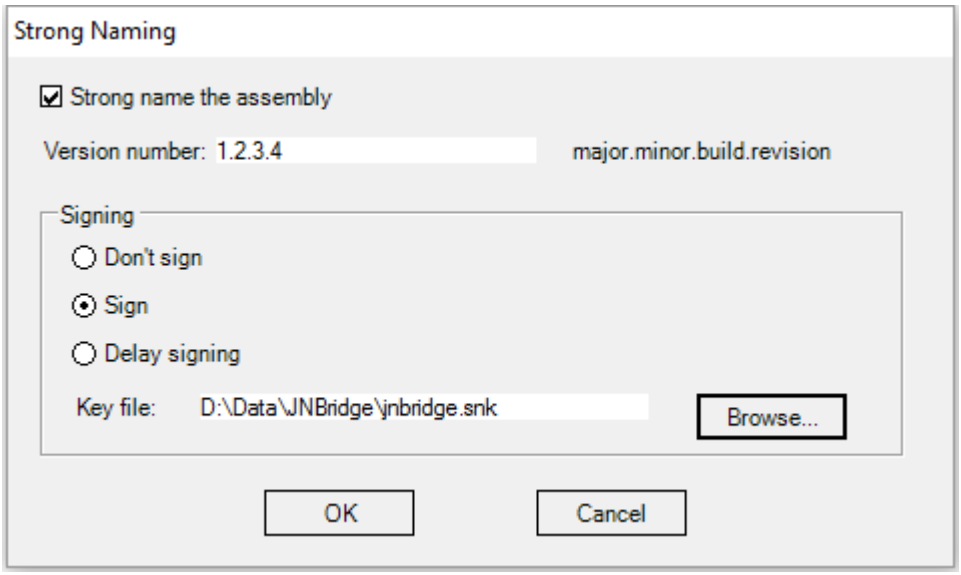


Figure 17. Strong Naming dialog box

The check box *Strong name the assembly* is used to indicate whether the assembly should be strong named. If the box is checked, the user must fill in the remaining options. The *Version number* field must contain a legal version number as described in the .NET Framework documentation for the class System.Version.

The user must choose a signing option. Choosing *Don't sign* will cause the proxy assembly to not be signed. Choosing *Sign* will cause the assembly to be signed, and choosing *Delay signing* will allocate space in the assembly so that the assembly can be signed later, according to the .NET delay signing process. In the event that *Sign* or *Delay signing* is chosen, the location of a Strong Name Key (.snk) file must be given.

Strong naming options are saved as part of the JNBProxy project (.jnb) files, and are restored when JNBProxy is used to open such a project file.

JNBProxy (Visual Studio plug-in)

The **JNBProxy** plug-in for Visual Studio 2005, 2008, 2010, 2012, 2013, 2015, 2017, and 2019, can be used to generate .NET proxies for Java classes.

Installing the Visual Studio plug-in (Visual Studio 2017, 2019 or 2022)

Up through Visual Studio 2015, the Visual Studio plug-in is installed automatically when JNBridgePro is installed in development mode. Starting with Visual Studio 2017, an additional step must be performed. In the JNBridgePro installation folder, locate the folder "VS plugin (2017 and later)." In that folder, you will find a file JNBridgePlugin2017.vsix. Assuming Visual Studio 2017, 2019 and/or 2022 is installed, double-click on this file in Windows Explorer. The Visual Studio extension manager will come up. When prompted, select the versions of Visual Studio in which you want the plug-in to be installed, then follow the remaining instructions. When the process is complete, the plug-in will be installed.

The plug-in packaged in the VSIX file will only work if JNBridgePro has already been installed in development mode *on that machine*, and if a valid development license has been deployed to that machine.

Initial invocation

The first time a JNBridgePro project is opened using the JNBProxy plug-in for Visual Studio (or the first time the JNBridgePro GUI-based proxy generation tool is used), its Java options must be configured. JNBProxy displays a copy of the Java Options window, as in Figure 18 below. (Note: you may reconfigure these values later via the **JNBridgePro→JNBridgePro Java Options…** menu item in Visual Studio, which brings up the same window. You may also configure these values using the **Tools→Options…** menu item in Visual Studio and navigating to the JNBridgePro tab.)

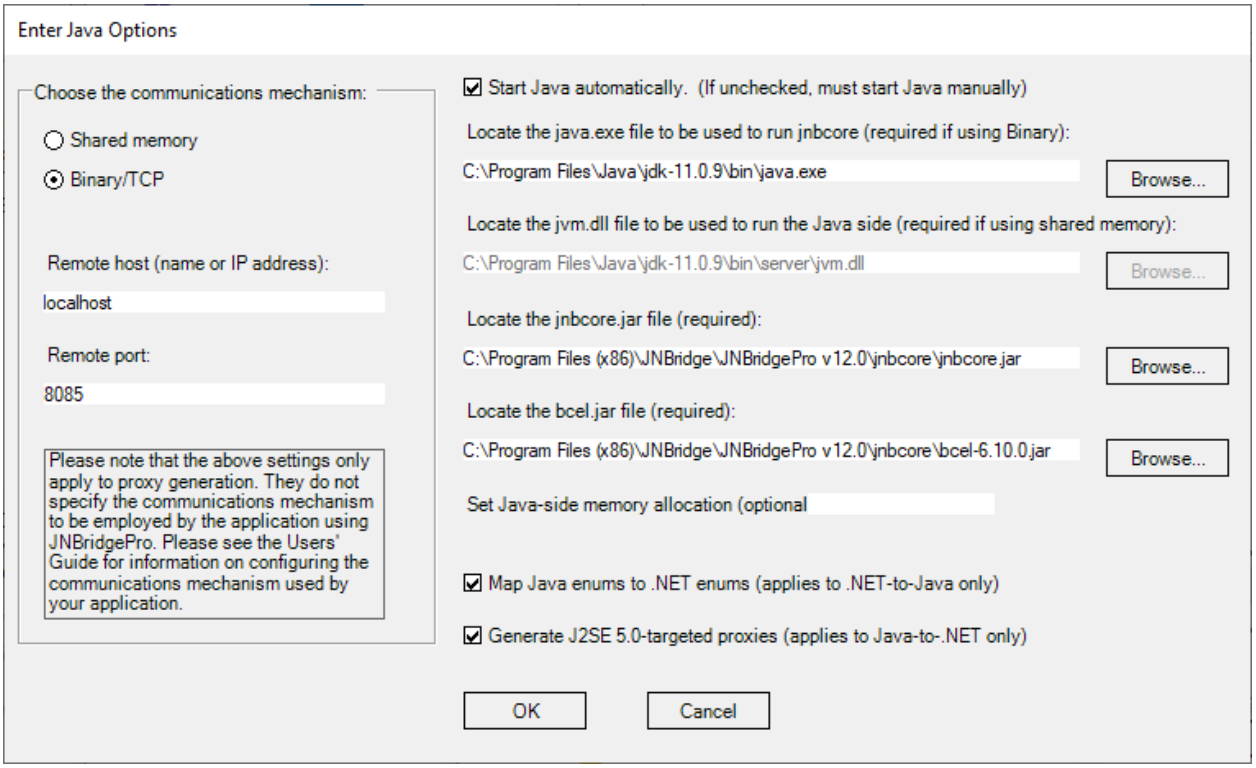


Figure 18: Java Options window

JNBProxy will attempt to fill in these options itself. Typically, there is no need to do anything further; the supplied choices will be sufficient. However, if JNBProxy cannot find a necessary file, or if unusual conditions apply, you may need to explicitly specify one or more of the options.

Two communications mechanisms are available for managing the communications between the .NET and Java sides: shared memory, binary/TCP. If you select binary/TCP, you must specify the host on which the Java side resides, and the port through which it will listen to requests from the .NET side. If you select shared memory, you must locate the file jvm.dll that implements the Java Virtual Machine that will be run in-process with the .NET-side code. Typically, jvm.dll is included as part of the JDK (Java Development Kit) or JRE (Java Runtime Environment) on which you will be running the Java side.

The default settings (binary/TCP, localhost, and port 8085) are typically sufficient and usually need not be changed.

Note: While shared memory is faster than binary/TCP, it has not been chosen as the default setting due to certain limitations in the Java Native Interface (JNI), which require that JNBProxy be exited and restarted whenever JNBProxy's classpath is changed.

Note: If shared memory is chosen while using the Visual Studio plug-in, then only one JNBridgePro project may appear in the currently open solution in Visual Studio. With binary/TCP, multiple JNBridgePro projects can appear in a Visual Studio solution.

If Java is to be started automatically, you must supply the location of the Java runtime (`java.exe`), along with the locations of the `jnbcore.jar`, `jnbcore.properties`, and the BCEL jar files. (Note that the BCEL jar file is only used in the *Java-to-.NET* direction, but its location must always be specified.)

If you wish to start Java manually, uncheck the "Start Java automatically" box.

Note: if the Java side is located on another machine, JNBProxy cannot start it automatically; it must be started manually.

Creating or opening a JNBridgePro project

To create a new JNBridgePro project from Visual Studio, select Create Project from the Start page, or select the **File→New→Project...** menu item, or right-click on the Solution node in the solution node in the Visual Studio solution explorer and select **Add→New Project...**. The New Project dialog will be displayed (Figure 19 and Figure 20). Note that there is a JNBridge entry under Project Types. Select the DotNetToJavaProxies template. You can choose the name and location of the project.

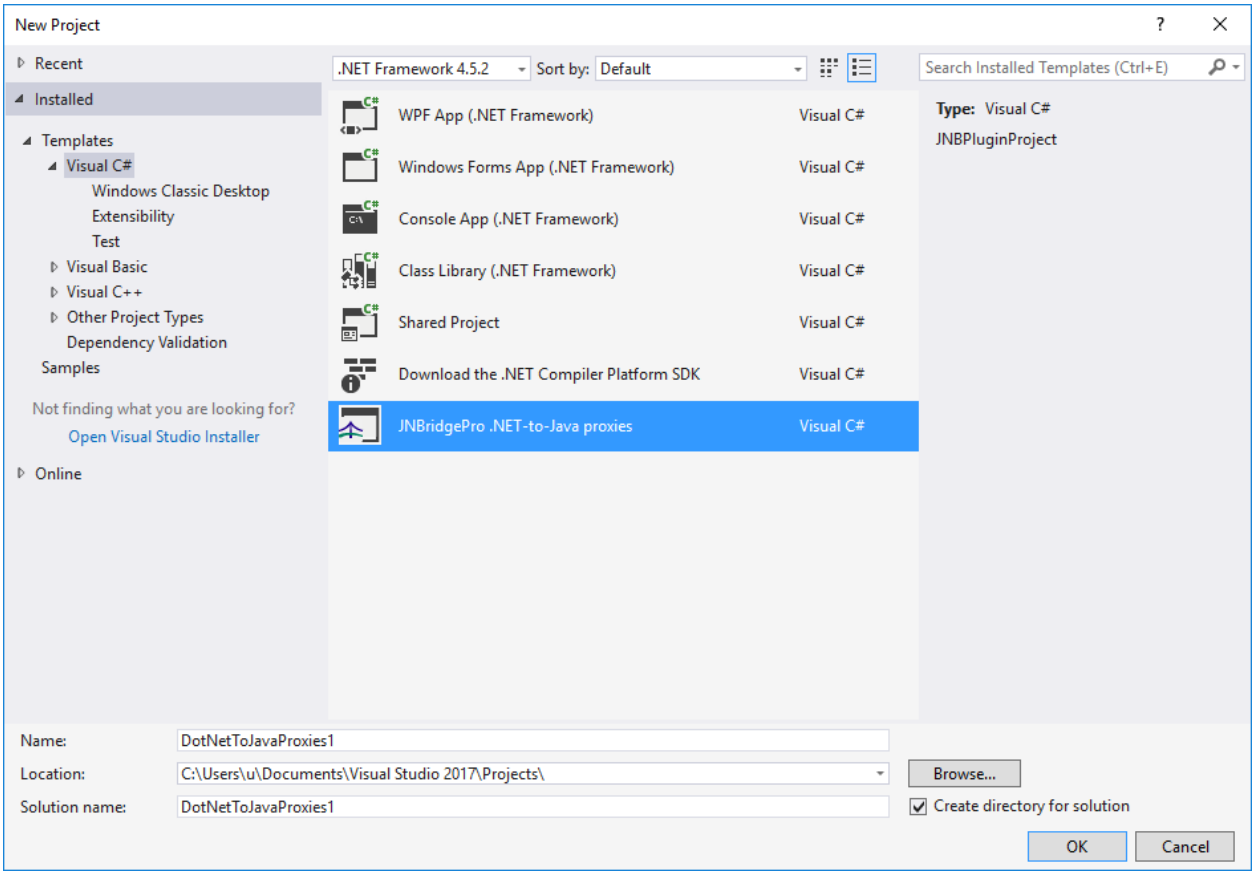


Figure 19. New Project dialog box (VS2017)

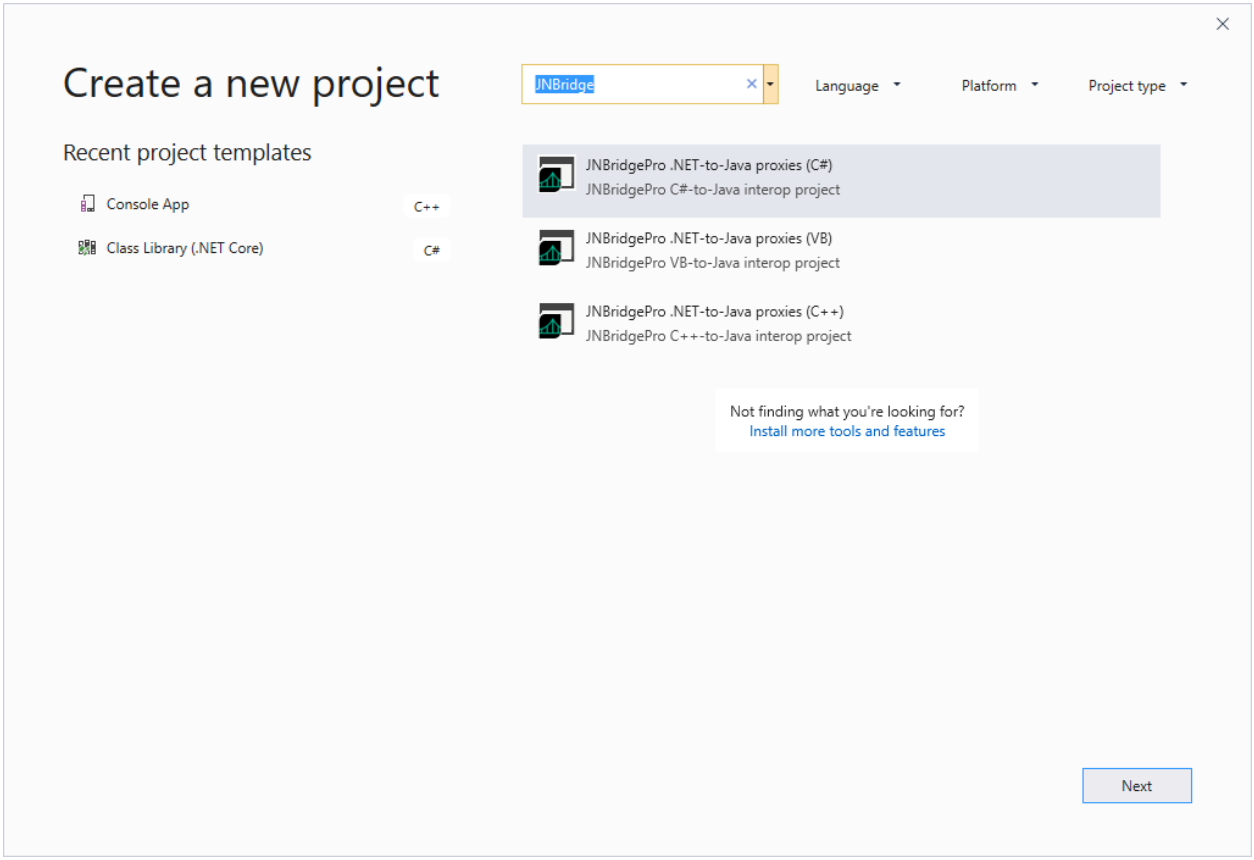


Figure 20. New Project dialog box (VS2019)

When the project is created, it is added to the current solution in the Visual Studio Solution Explorer (Figure 21). Note that there is a Project node, representing the JNBridgePro Visual Studio Project (.jnbproj) file, and a JNBridgePro Proxy Generation Project (.jnb) file. The .jnb file is the same as the .jnb file created by the JNBridgePro GUI-based proxy generation tool and can also be read and modified by that tool.

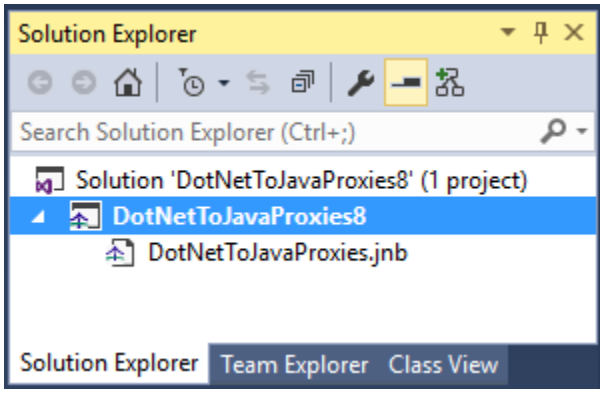


Figure 21. New JNBridgePro project in Solution Explorer

It is also possible to open an existing JNBridgePro project, or to add an existing JNBridgePro project to your current solution. To do so, select Open Project… in the Visual Studio Start page, or select the **File→Open→Project/Solution...** menu item, or right-click on the solution node in the Solution Explorer and select **Add→Existing Project...**, then navigate to the desired .jnbproj file and select it.

Note: The platform pull-down in the New Project dialog box does not influence the generated proxies' target platform. Changing it will have no effect on the generated proxies. To change the generated proxies' target platform, see the section "Multi-targeting," below.

Opening a JNBridgePro project

To open a JNBridgePro project, simply double-click on its .jnb file node in the Solution Explorer, or select the **File→Open→File...** menu item. A JNBProxy editor will open (Figure 22).

GUI layout

Figure 22 shows a JNBProxy editor immediately after it has been opened from a newly created .jnb file. Within the editor's window are three panes: the *environment pane*, the *exposed proxies pane*, and the *signature pane*. There is also a new *JNBridge* sub-pane within Visual Studio's Output pane.

The environment pane displays the Java classes of which JNBProxy is aware, and for which proxies can be generated. The exposed proxy pane shows those Java classes for which proxies will be generated. The signature pane shows information on the methods and fields offered by a Java class, as well as other information on the class. The output pane displays diagnostic and informative messages generated during the operation of the tool. You may drag the splitter controls to resize the panes.

Also note that there is a JNBridgePro tool bar that can be displayed with other tool bars in Visual Studio.

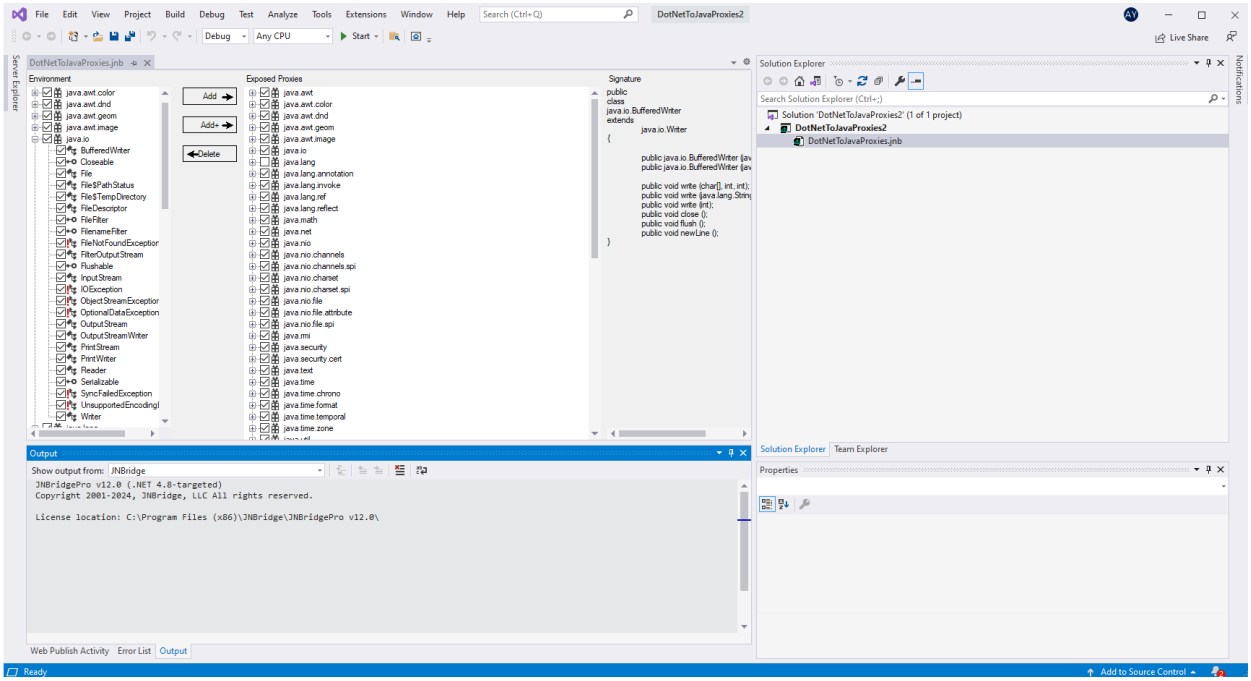


Figure 22. JNBridgePro proxy generation editor

Note that there can be multiple JNBProxy editors open at the same time in Visual Studio, each corresponding to a different .jnb file.

Process for generating proxies

Generating proxies takes three steps:

- Add classes to the environment for which you might want to generate proxies.
- Select classes in the environment for which proxies are to be generated and add them to the exposed proxies list.
- Build the proxies into a .NET assembly.

Adding classes to the environment

The first step in generating proxies is to load candidate classes into the environment. Think of the environment as a palette from which one can choose the proxies that are actually generated. Classes can be loaded into the environment in two ways: from a jar file, and as specific classes found along the Java classpath.

To load classes from a jar file, click on the menu item **JNBridgePro→Add classes from JAR file…**, or right-click on the .jnb file node in the Solution Explorer and select **Add classes from JAR file…**. A dialog box will appear allowing the user to locate one or more jar files whose classes will be loaded. Opening a jar file will cause all the classes inside of it to be added to the environment pane. The progress of the operation is shown in the output pane. The operation can be terminated before completion by clicking on the Stop button located on the GUI's tool bar.

Note: any jar file from which classes are to be loaded must be in the Java classpath. To edit the classpath, see "Setting or modifying the Java classpath," below.

Note: The results of operations terminated prematurely by clicking the stop button may be inconsistent and should not be used. These partial results can be removed by performing an Undo operation.

To load specific classes from the classpath, click on the menu item **JNBridgePro→Add classes from classpath…**, or right-click on the .jnb file node in the Solution Explorer and select **Add classes from classpath…**. When this item is selected, the Add Classes dialog box is displayed (Figure 23).

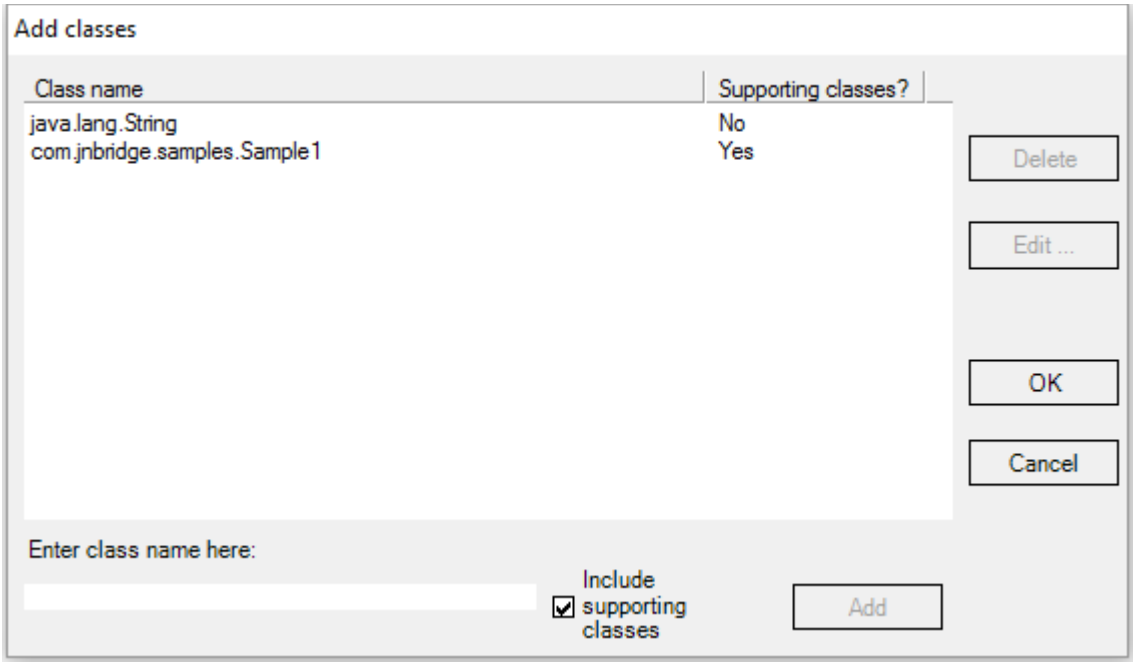


Figure 23. Add classes dialog box

To specify a class to be added, type the fully-qualified class name into the text box at the bottom of the dialog box. (Note that, as you type, the interface will provide class name suggestions based on what is available and what you have typed so far.) Leave checked the "Include supporting classes" check box to indicate that all classes supporting the specified class should be automatically added. (See *Supporting classes*, below, for more information on supporting classes.) Then, click the Add button to add the class to the list of classes to be loaded into the environment. You may delete a class from this list by selecting the class in the list and clicking on the Delete button. You may edit the information for the class by selecting the class and clicking on the Edit button or by double-clicking on the class. When you are done specifying classes, click the OK button to add the classes to the environment. This process may take a few minutes, especially if any of the classes in the list must also have their supporting classes loaded. The operation can be terminated before completion by clicking on the Stop button located on the GUI's tool bar.

Also note that, in addition to specifying the fully-qualified name of a class to be added, you can also use a trailing '*' as a wildcard, to indicate that all classes in the classpath that match begin with the string that precedes the asterisk should be added. For example, supplying 'java.util.*' indicates that all the classes in the java.util.* package should be added.

Note: If you are doing Java-.NET co-development, and a Java class is changed after the class has been loaded into the environment (e.g., a method or field has been added or removed, or the signature changed), the class can be updated in the environment simply by loading it again.

Note: Any class or supporting class that is added must be in the classpath or a member of one of the standard java.* packages.

Note: Proxies for java.lang.Object and java.lang.Class are always generated and added to the environment even if they are not explicitly requested or implicitly requested by checking "Include supporting classes."

Note: The results of operations terminated prematurely by clicking the stop button may be inconsistent and should not be used. These partial results can be removed by performing an Undo operation.

Selecting proxies to be generated

Once classes have been loaded into the environment, they are displayed in the environment pane, which is a tree view listing all the classes loaded into the environment grouped by package. Next to each item in the tree view is an icon indicating whether the item is a package, an interface, and exception, or some other class. (See Figure 24)

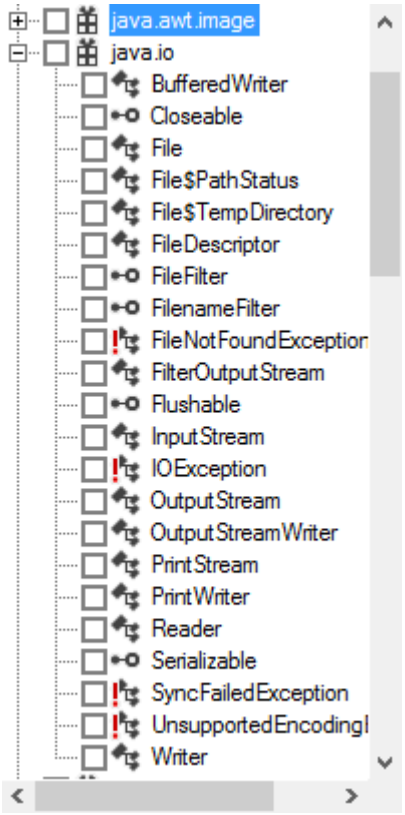


Figure 24. Environment pane

To select a class for proxy generation, check the class's entry in the environment pane by clicking on the check box next to the name. To select all the classes in a package, check the package's entry by clicking on its check box. Once a set of classes has been checked, add them to the set of exposed proxies by clicking on the **Add** button.

Alternatively, you may click on the **Add+** button to add the checked classes and all supporting classes. For a discussion on supporting classes, see the *Supporting Classes* section later in this guide. The checked classes and all supporting classes (if **Add+** was clicked) will appear in the exposed proxies pane, which is a tree view similar to the environment pane.

Note: use of* Add+ *may take a few minutes for the operation to complete. The operation can be terminated before completion by clicking on the Stop button located on the GUI's tool bar. The results of operations terminated prematurely by clicking the stop button may be inconsistent and should not be used. These partial results can be removed by performing an Undo operation.

To selectively remove classes from the exposed proxies pane so that proxies are not generated, check the entries in the exposed proxies pane for all classes or packages to be removed (by clicking on the check boxes next to the class or package names), then click on the **Delete** button. The checked items will be removed from the exposed proxies pane.

As a convenience, the JNBridgePro menu contains items to check or uncheck all the items in the environment or exposed proxy panes.

Add, **Add+**, and **Delete** operations may be repeatedly performed until the exact set of proxies to be generated appears in the exposed proxies pane.

The most recent **Add**, **Add+**, and **Delete** operations may be undone and then redone using the **Edit→Undo** and **Edit→Redo** menu items.

Note: Proxies for java.lang.Object and java.lang.Class will always be added to the exposed proxies pane when the Add button is clicked, even if they have not been checked in the environment pane. Also, java.lang.Object and java.lang.Class cannot be checked in the exposed proxies pane, and therefore cannot be deleted from that pane. The reason for this behavior is to ensure that proxies for Object and Class are always generated.

Designating proxies as reference or value

Before generating proxies for the classes listed in the exposed proxies pane, the user can designate which proxies should be reference and which should be value. (See the section "Reference and value proxies," below, for more information on the distinction between reference and value proxies.) The default proxy type is reference; if the user wants all proxies to be reference, nothing additional need be done.

To set a proxy's type (i.e., to reference or any of the three styles of value), position the cursor over the proxy class to be changed (in the exposed proxies pane), and right-click on the class. A pop-up menu will appear. Choose the desired proxy type. After the proxy type is selected, the proxy class will be color coded to indicate its type (black for reference, blue for value (public/protected fields style), red for by-value (JavaBean style), or green for by-value (mapped).

To set the types of multiple proxy classes at the same time, click on the **JNBridgePro→Pass by Reference / Value…** menu item. The Pass by Reference / Value dialog box is displayed (Figure 25). All proxy classes in the exposed proxies pane are listed in this dialog box. To change the types of one or more classes, select those classes (left-click on a class to select it, and shift-left-click or ctrl-left-click to do multi-selects), then click on the Reference, Value (Public/protected fields), Value (JavaBeans) or Value (Mapped) button to associate the selected classes to the desired type. When done, click on the OK button to actually set the classes to the chosen types and dismiss the dialog box, click on the Apply button to set the classes without dismissing the dialog box, or click on the Cancel button to discard all changes.

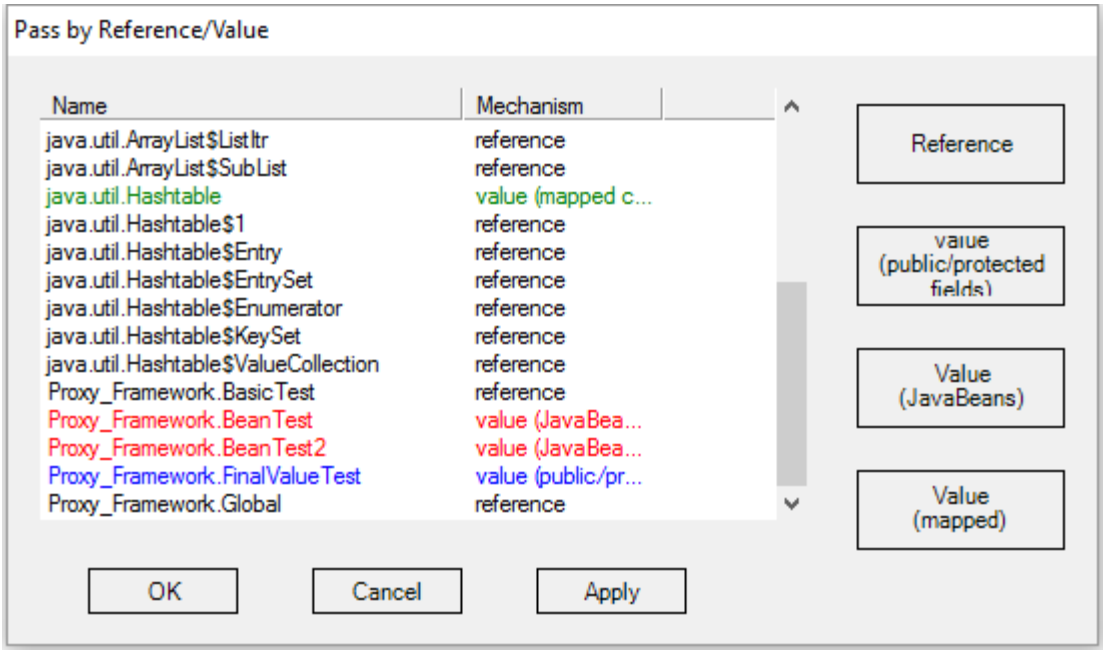


Figure 25. Pass by Reference/Value dialog box

Designating proxies as transaction-enabled (formerly "thread-true")

Before generating proxies for the classes listed in the exposed proxies pane, the user can designate which proxies should be *transaction-enabled*. (See the section "Transaction-enabled classes and support for transactions," below, for more information on transaction-enabled classes.) The default proxy type is not transaction-enabled; if the user wants all proxies to be non-transaction-enabled, nothing additional need be done.

To set a proxy as transaction-enabled, position the cursor over the proxy class to be changed (in the exposed proxies pane), and right-click on the class. A pop-up menu will appear. Select the "transaction enabled" menu item so that it is checked. To remove the transaction-enabled property from a proxy class, perform the same operation to uncheck the transaction-enabled menu item. It is currently not possible to make multiple classes transaction-enabled at the same time.

Users should use the transaction-enabled property sparingly, as it incurs a performance penalty. See the section "Transaction-enabled classes and support for transactions" for more information.

Generating the proxies

Once classes have been selected for proxy generation and have been moved into the exposed proxies pane, a .NET assembly with the proxies can be generated by selecting a Build or Rebuild operation for either the entire solution or the current JNBridgePro project. The assembly will be written to the project's Debug or Release directory (depending on the current configuration), and its name will be identical to the name of the project's .jnb file.

It is also possible to generate the proxy assembly by clicking on the **JNBridgePro→Build...** menu item. In this case, a dialog box will be displayed that allows the user to specify the location and name of the .NET assembly that will contain the generated proxies.

When proxies are generated, if one or more proxy classes are value, a consistency check will be performed before the proxies are generated (see "Reference and value proxies," below), and the results will be reported to the user.

Note: The Build operation may take a few minutes for the operation to complete. The operation can be terminated before completion by clicking on the Stop button located on the GUI's tool bar. The results of operations terminated prematurely by clicking the stop button may be inconsistent and should not be used.

However the proxies are built, information concerning the generation of the proxies is displayed in Visual Studio's Output pane, in both the Build and JNBridge sub-panes. If the build was unsuccessful, information describing the errors will be found there.

Generating and using proxies as part of a larger build

It is possible to generate and use proxies as part of a larger build operation. To do so, simply right-click on the node in the Solution Explorer representing the project that will use the proxies, select **Add Reference…**, and select the Projects tab. Then select the project representing the generated proxies. One must also add a references to jnbshare.dll from the JNBridgePro installation folder. Then, when one performs a build, the referencing project will use the proxy assembly generated by the JNBridgePro project. If the JNBridgePro project is out of date or has not yet been built, it will automatically be built or rebuilt before being used.

Note: In addition to adding jnbshare.dll as a reference in your project, also add the jnbsharedmem_x86.dll and/or jnbsharedmem_x64.dll to your project. Right-click on the node representing the project, select **Add→Existing Item…,** and select either or both of the jnbsharedmem dlls, depending on whether you are creating an x86, x64, or Any CPU project. For each added item, set its property to Copy Always, so that a copy of the dll is always included in the build.

Information converning the generation of the proxies is displayed in Visual Studio's Output pane, in both the Build and JNBridge sub-panes. If the build was unsuccessful, information describing the errors will be found there.

Saving projects

The current contents of the proxy generation editor is automatically saved to the .jnb file when the JNBridgePro project is built. The contents of the editor can be explicitly saved by selecting one of the **Save** items under the **File** menu. In addition, if the proxy generation editor is closed, and its contents has been modified since it was last saved, the user is offered the opportunity to save it.

The saved .jnb file can also be read and edited by the standalone JNBProxy GUI-based proxy generation tool.

Exporting a class list

Selecting the **JNBridgePro→Export classlist…** menu item will cause a text file to be generated that lists the classes in the Exposed Proxies pane, plus information on whether the classes are by-reference or by-value, and whether the classes are transaction-enabled. The exported class list is in the correct format to be used as input with the '/f' option of the command-line version of JNBProxy. (See the section "Using JNBProxy (command-line version)").

Generating a command-line script

In certain cases, it will be useful or necessary to incorporate proxy generation into an automated build process. The command-line version of the proxy generator (see below) is intended for these situations. To simplify the creation of command-line scripts, JNBProxy can be used to automatically generate a script for the current project. Select the **JNBridgePro→Generate command-line script…** menu item to generate a .bat file that can be used to invoke the command-line proxy generator. When generating a command-line script, the user will be prompted to supply the name of the .bat file. At the same time, a class list will be generated (see "Exporting a class list," above). If the .bat file is named *myFile.bat*, the class list will be named *myFile_classlist.txt*, unless a file of that name already exists, in which case the user will be asked whether it should be overwritten, or whether the file should have a different name.

Note that the .bat file can, and in many cases should, be edited, particularly if it will be run in a different location from that to which it was written. For information on the command-line options and how they can be edited, please see the section "Using JNBProxy (command-line version)," below.

Examining class signatures

JNBProxy displays information about a Java class so that a user can determine whether it contains properties of interest to the user. If a class item in either the environment or exposed proxies pane is selected, its information is displayed in the signatures pane. The information includes the class's name, superclass, attributes, interfaces, fields, and method signatures (Figure 26).

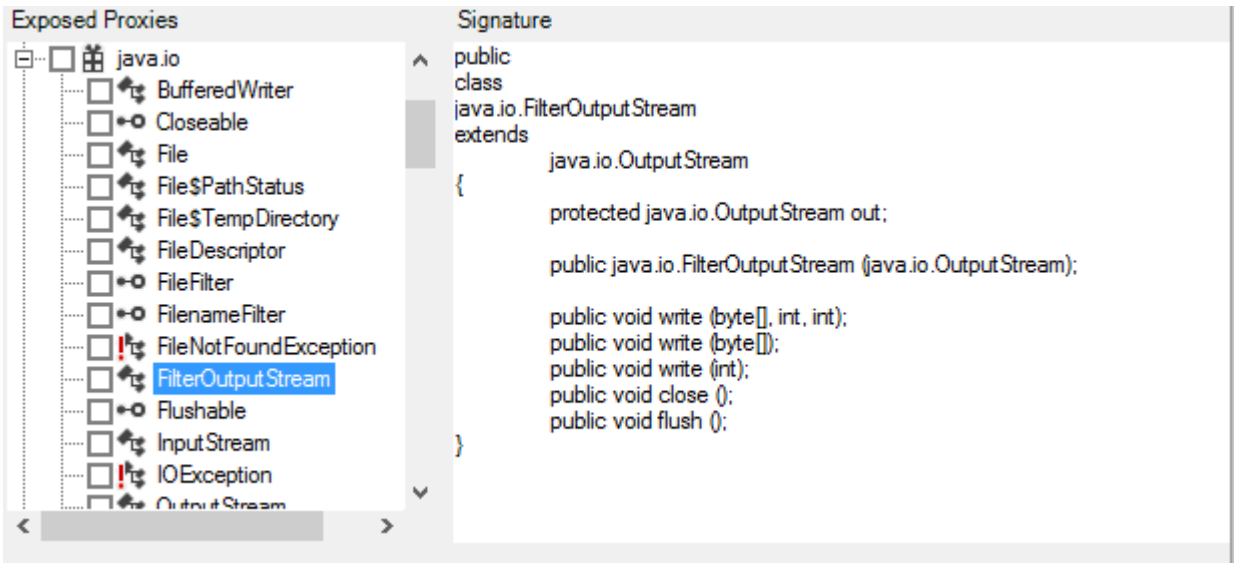


Figure 26. Selected item in exposed proxy pane and corresponding information in signature pane

Searching for classes in the environment or exposed proxy panes

To find a class among a large number of classes in the environment or exposed proxy panes, use the Find facility available by clicking on the **Edit→Find and Replace→Quick Find (Ctrl+F)** menu item, which displays a Find window (Figure 27). The Find window offers a variety of options, including the ability to search the environment or exposed proxy panes, to search down or up, to look for exact

whole-word matches, and to perform case-dependent matches. To repeat a search, the user should use **Find Next (F3)** key.

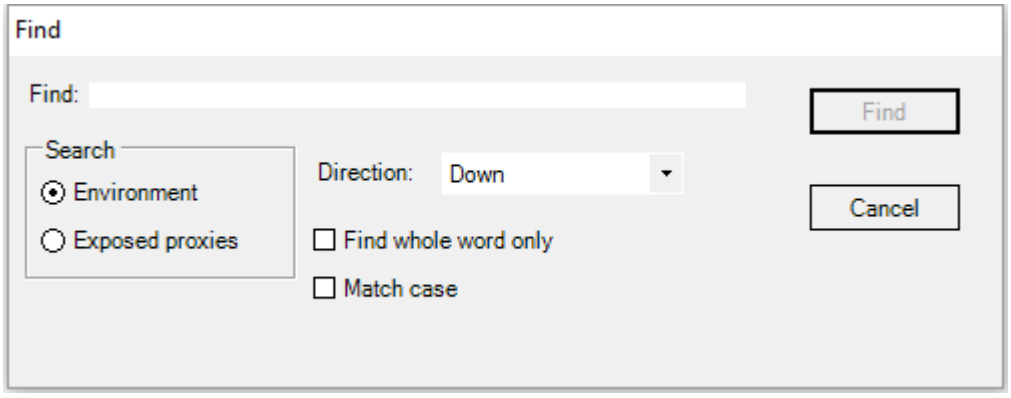


Figure 27. Find window.

Modifying tree view properties

To change the ways that the environment and exposed proxy panes are displayed, click on the **JNBridgePro→Tree View Options...** menu item. When this option is selected, a dialog box is displayed that allows the user, for either the environment or exposed proxy pane, to show or hide the icons in the pane, and to completely expand or completely collapse the tree view in the pane (Figure 28).

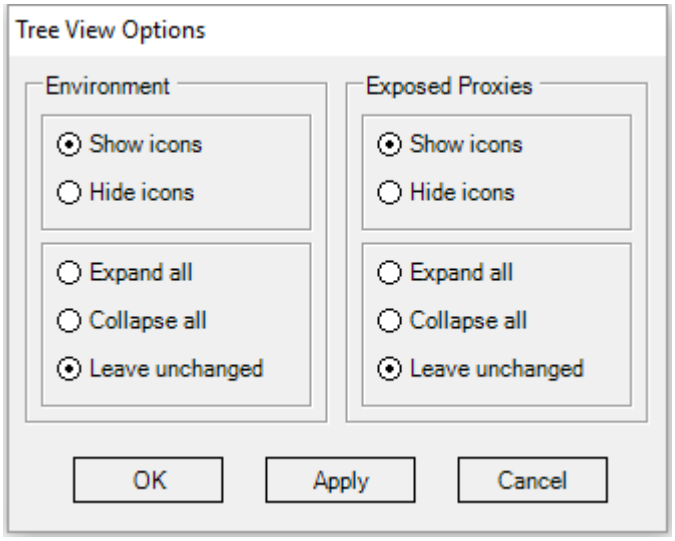


Figure 28. Tree view options window

Refreshing the display

To refresh the contents of the three editor panes (environment, exposed proxy, and signature) in the proxy generation editor window, select the **JNBridgePro→Refresh** menu item.

Setting Java startup options

The user may choose for the proxy generator to automatically start the Java side, or the user may start it manually. (See *Starting Java manually* for more information.) To control whether or not Java is started automatically, and to specify the location of various executables and libraries, select the **JNBridgePro→JNBridgePro Java Options…** menu item (Figure 29), or select the JNBridge section in the **Tools→Options…** dialog.

Two communications mechanisms are available for managing the communications between the .NET and Java sides: shared memory, binary/TCP. If you select binary/TCP, you must specify the host on which the Java side resides, and the port through which it will listen to requests from the .NET side. If you select shared memory, you must locate the file jvm.dll that implements the Java Virtual Machine that will be run in-process with the .NET-side code. Typically, jvm.dll is included as part of the JDK (Java Development Kit) or JRE (Java Runtime Environment) on which you will be running the Java side.

The default settings (binary/TCP, localhost, and port 8085) are typically sufficient and usually need not be changed. **Note that these settings only apply to proxy generation, not to the way the Java and .NET sides will communicate in the application under development. To configure the communication in the application you are developing, see the section "Using Proxies with JNBridgePro."**

Note: While shared memory is faster than binary/TCP, it has not been chosen as the default setting due to certain limitations in the Java Native Interface (JNI), which require that the proxy generation plug-in (and, therefore, Visual Studio) be exited and restarted whenever JNBProxy's classpath is changed.

To cause Java to be started automatically by the JNBProxy plug-in, check the check box at the top of the Java options window. Leave the box unchecked if you wish to start Java manually.

If Java is to be started automatically, you must supply the location of the Java runtime (`java.exe`), along with the locations of the `jnbcore.jar` and `jnbcore.properties` and the BCEL jar files. (the BCEL jar file is only used in the Java-to-.NET direction, but its location must always be specified.)

If you wish to start Java manually, uncheck the "Start Java automatically" box.

Note: if the Java side is located on another machine, JNBProxy cannot start it automatically; it must be started manually.

If the user's Windows account contains privileges allowing it to write new settings to the registry, the Java Options settings will be saved from one invocation of JNBProxy to the next; if the account does not have such privileges, the settings will not be saved.

Adjusting the Java-side memory allocation during proxy generation

When generating proxies for a very large number of proxies (for example, all the classes in a very large jar file), it is possible that the Java side may run out of memory, and proxy generation will fail. When this happens, one can avoid the problem by specifying a larger Java-side memory allocation to be used during proxy generation.

To adjust the Java-side memory allocation, bring up the Java Options dialog box by selecting the **JNBridgePro→JNBridgePro Java Options…** menu item. The Java Options window (Figure 29) will be displayed. One can specify a new Java-side memory allocation by entering it in the box labeled "Set Java-side memory allocation." The value entered there is the new Java-side memory allocation in bytes, and may be either a number that is a multiple of 1024, a number followed by "m" or "M"

(denoting megabytes), or a number followed by "k" or "K" (denoting kilobytes). If the entry is left blank, the default memory allocation for the current Java runtime environment will be used.

Note: The Java-side memory allocation setting only applies to proxy generation. It does not affect subsequent use of the proxies. To specify a Java-side memory allocation to be used during the running application, you need to supply a –Xmx option to the Java side.

Note: If the "Start Java automatically" checkbox is unchecked, the Java-side memory allocation option will be ignored.

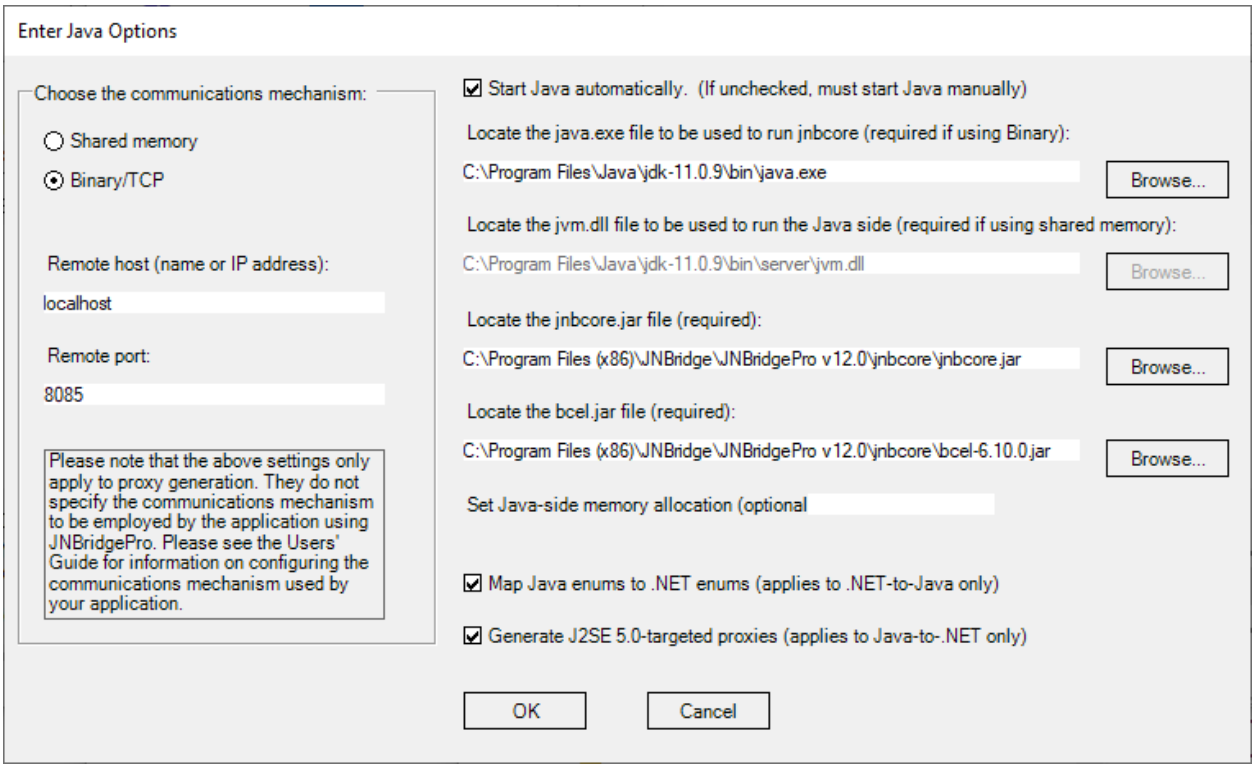


Figure 29. Java Options window.

Proxying Java enums

Originally, Java enum types were proxied in the same way as any other Java type. The proxied enums would inherit from the proxy of java.lang.Enum, and all of the enum's associated methods could be accessed through the proxies. Later versions of JNBridgePro mapped Java enums to .NET enums during proxy generation. While this made accessing the enumerated values much more efficient, it resulted in loss of access to the methods in the underlying Java enums, and it made it impossible to proxy a Java class that was a subclass of an enum (something that is legal in Java), because a subclass of a .NET enum is not permitted. It also made it difficult for some users to upgrade to newer versions of JNBridgePro.

With the current version of JNBridgePro, the user can control the way in which Java enums are proxied. A checkbox in the Java Options dialog box (Figure 29) labeled "Map Java enums to .NET enums" controls this behavior. The default is for the box to be checked, in which case the Java enums are mapped to .NET enums. To simply proxy Java enums in the same way as other classes, uncheck this box before generating the proxies.

This check box only applies to .NET-to-Java projects. It is ignored in Java-to-.NET projects.

Setting or modifying the Java classpath

If Java is started automatically, the Java classpath should be set. Do this by selecting the **JNBridgePro→Edit Classpath...** menu item. This will call up the Edit Classpath window (Figure 30).

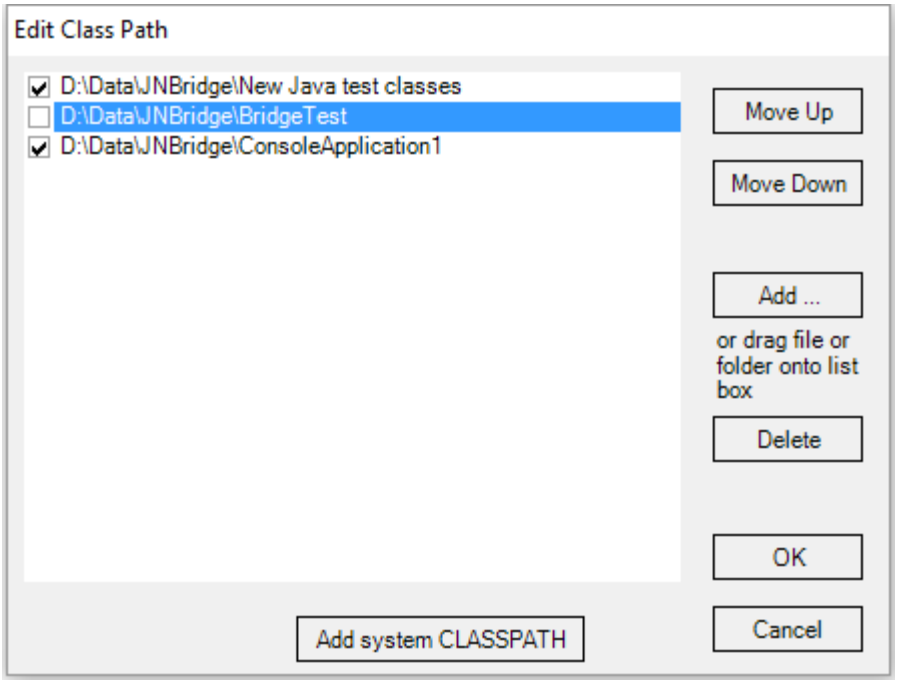


Figure 30. Edit Classpath window.

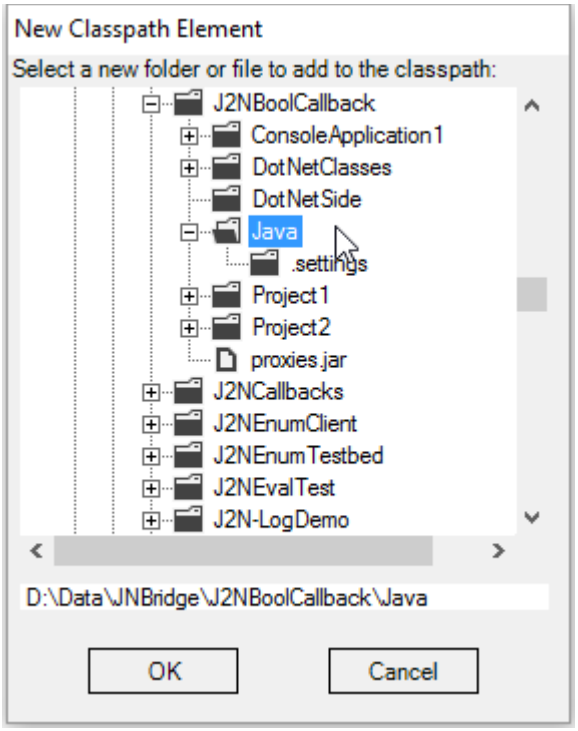


Figure 31. New Classpath Element window.

The classpath can be rearranged by selecting a classpath element and clicking on the **Move Up** or **Move Down** buttons, or by selecting an element and clicking on the **Delete** button. The files and folders in the system CLASSPATH environment variable can be added to the classpath list by clicking on the **Add system CLASSPATH** button. Note that only checked elements will be added to the Java classpath when the dialog is dismissed.

To add a folder or jar file to the classpath, simply drag and drop one or more files or folders onto the Edit Classpath form. If you drag a folder, you will be given the choice of simply adding the folder to the classpath (useful if you are developing Java using .class files), or of adding all the jar files contained in the folder to the classpath. You are also given the option of telling the proxy generation tool to make the same choice for all subsequent adds of folders. (If you select this option, you can always change that decision by selecting **Tools→Options…**, then selecting the **JNBridgePro→Proxy generation** tab and selecting "Always ask" as the "Folder drag-and-drop" action.

You can also add a folder or jar file by clicking on the **Add…** button. This causes a New Classpath Element window to be displayed. In this window, the user can navigate to the desired folders or jar files, or can enter a file path directly (Figure 31). The New Classpath Element window supports multiple selection - multiple folders and/or jar files may be selected by ctrl-clicking, while a range of folders and/or jar files may be selected by shift-clicking. Clicking on the **OK** button will cause the indicated folders or files to be added to the Edit Classpath window.

Note: You can also type a directory or file path directly into the New Classpath Element dialog box. This path can be an absolute path, or a UNC path to a shared network folder (e.g., \\MachineName\FolderName\FileName.jar).

If the classpath is modified while the Java side is running, and automatic Java startup is set, the Java side is automatically halted and restarted.

Note: if the Java side is started up manually, changing the JNBProxy classpath will have no effect.

Note: If Java is started up automatically, terminating JNBProxy with an external kill may leave a stray Java process running that you will have to terminate as well.

Note: If shared-memory communication between the .NET and Java sides is used, it will be necessary to exit and restart Visual Studio whenever the classpath is changed. This is due to limitations in the Java Native Interface (JNI), which is used by the shared-memory channel.

Strong naming proxy assemblies

.NET supports the notion of *strong naming* of assemblies, which includes the assignment of version numbers as well as digitally signing the assemblies. For more details on strong naming of assemblies, see the .NET Framework documentation.

To strong name a proxy assembly, the strong naming options must be set before the proxy assembly is built. To set strong naming options, select the **JNBridgePro→JNBridgePro Strong Naming…** menu item. The Strong Naming dialog box will be displayed (Figure 32).

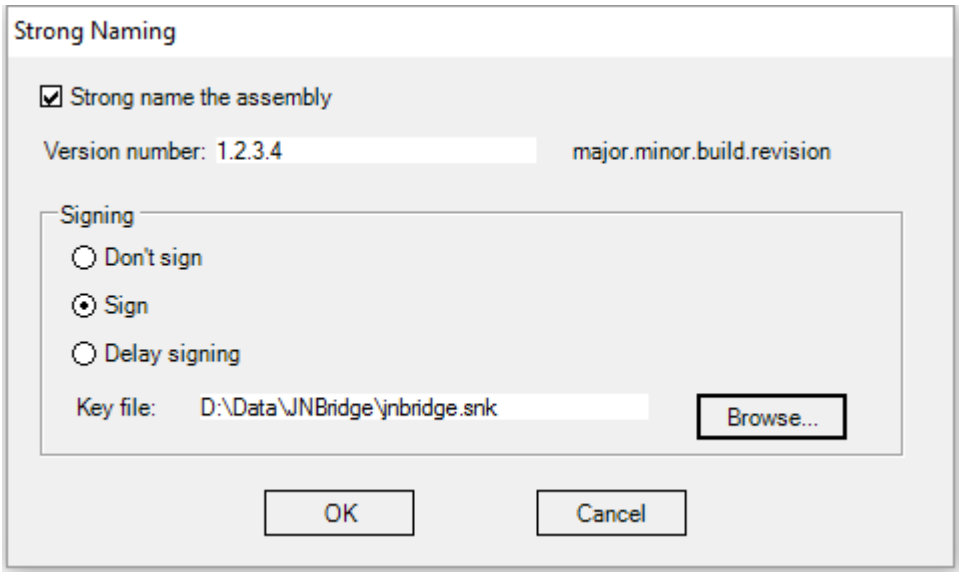


Figure 32. Strong Naming dialog box

The check box *Strong name the assembly* is used to indicate whether the assembly should be strong named. If the box is checked, the user must fill in the remaining options. The *Version number* field must contain a legal version number as described in the .NET Framework documentation for the class System.Version.

The user must choose a signing option. Choosing *Don't sign* will cause the proxy assembly to not be signed. Choosing *Sign* will cause the assembly to be signed, and choosing *Delay signing* will allocate space in the assembly so that the assembly can be signed later, according to the .NET delay signing process. In the event that *Sign* or *Delay signing* is chosen, the location of a Strong Name Key (.snk) file must be given.

Strong naming options are saved as part of the.jnb file, and are restored when the proxy generation editor is used to open such a project file.

Multi-targeting (Visual Studio 2010 and later)

The JNBridgePro plug-in for Visual Studio 2010, 2012, 2013, 2015, 2017, and 2019 (but not Visual Studio 2005 or 2008) offers the ability to choose whether the generated proxy assembly is targeted toward .NET Framework 2.0/3.0/3.5 (which all use the .NET 2.0 runtime), or .NET Framework 4.0/4.5/4.6/4.7/4.8 (both the full framework and the client profile).

To choose the targeted platform, right-click on the appropriate JNBridgePro interoperability project in Visual Studio's Solution Explorer and select "Properties." A property page for the project will open (Figure 33). One property on that page will be labeled "Target Platform" and will have a pull-down. Using the pull-down, select the desired target platform, then dismiss the property page by clicking on the OK button. When you build the project, the proxy assembly will be targeted toward the selected platform.

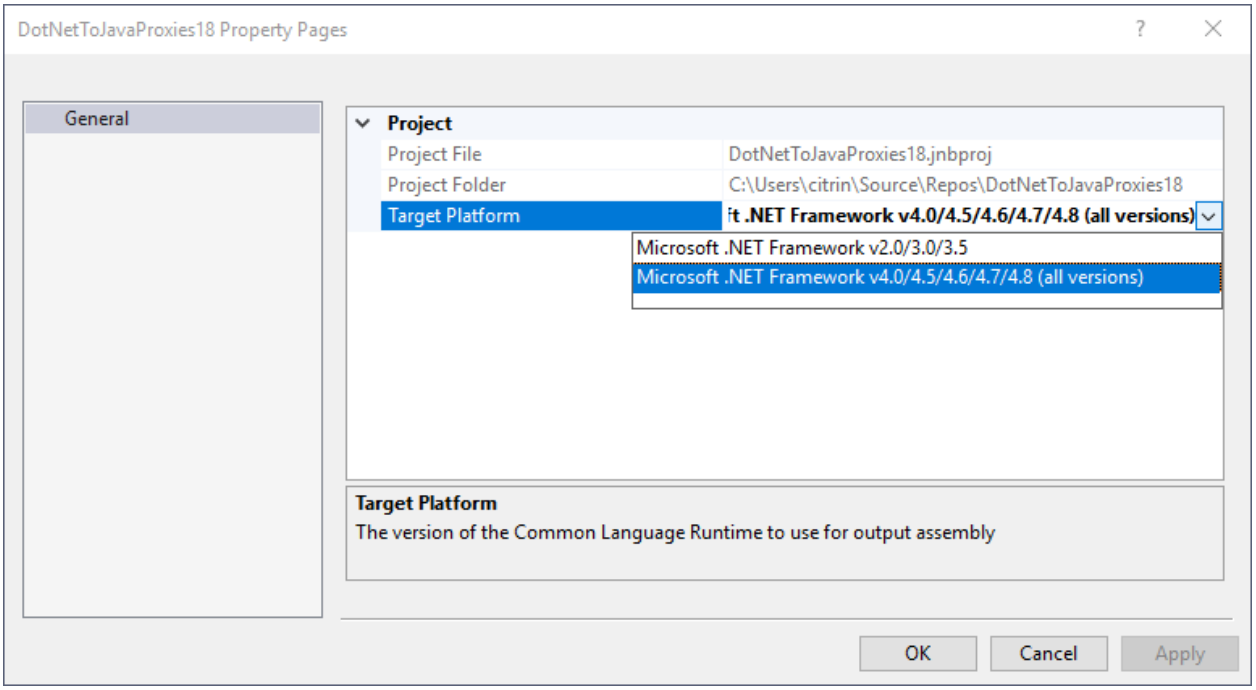


Figure 33. Choosing a target platform

Note: To change the target platform, you must change the pull-down in the property page above. Changing the platform pull-down in the New Project dialog box has no effect on multi-targeting.

Using JNBProxy (command-line version)

Use the **JNBProxy** command-line application (jnbproxy.exe) to generate .NET proxies for Java classes. Unless the /nj option is used (see below) and the Java side is started manually (or shared-memory communication is used), a copy of java.exe (or the folder in which it resides) must be found in the system execution path. If it is not, and java.exe is not found, an error will be reported and JNBProxy will terminate.

For the command-line version of jnbproxy to be used in a .NET-to-Java direction, the /pd option (see below) should be **n2j**.

JNBProxy is used from the command line as follows:

jnbproxy <options> *classes*...

classes… A space-separated sequences of fully qualified Java class names (e.g., `java.lang.String`) for which proxies should be generated. Note that proxies for `java.lang.Object` and `java.lang.Class` are always generated, even if they are not listed in the class list.

Options

/bp *bcelpath* BCEL classpath. *bcelpath* is the file path locating the folder containing the file `bcel-6.n.m.jar`. If left out, the environment variable CLASSPATH is used to locate `bcel-6.n.m.jar`. If using shared-memory communications, this option must be supplied.

/cp *classpath* Java classpath. *classpath* is a semicolon-separated series of file paths of folders and jar files containing the Java classes with which the .NET classes will communicate. If left out, the environment variable CLASSPATH will be used. In addition to the classes in the supplied or default classpath, the classes in the core Java API (the java.* namespaces) will be available.

/dme Don't map Java enums to .NET enums. When this option is included, Java enums are proxied in the same way as ordinary Java classes. When this option is omitted, Java enums are mapped to .NET enums. (Only applies to .NET-to-Java projects. When used in a Java-to-.NET project, this option is ignored.)

/d *directory* Directory. Write the generated proxies to an assembly in the specified *directory*. If left out, the default is the current execution directory.

/f *classfile* Read from file. Read classes for which proxies are to be generated from the specified text file rather than from the command line. The purpose of this option is to allow users to generate large numbers of proxies where it would be impractical to type them onto the command line. The file must consist of a list of fully-qualified class names, one per line. Optionally, the class name can be followed by white space (one or more spaces and/or tabs), followed by the letter "r" (denoting that the class should be a reference proxy – see "Reference and Value Proxies," below), "f" (denoting that the class should be a public/protected fields-style value proxy), "j" (denoting that the class should be a Java Beans-style value proxy), or "m" (denoting that the class should be a directly-mapped value proxy). In addition to the above, the class name can also be followed by the letter "t", indicating that the class should be *transaction-enabled* (see "Transaction-enabled classes and support for transactions," below). Any other trailing letter, or no letter, results in the generation of a reference proxy. If the /f option is used, any classes listed on the command line will be ignored.

/h Help. List the options and usage information. All other options and arguments will be ignored. Simply typing the command *jnbproxy* with no options or arguments will result in the same information

/host *hostname* Host. The host on which the Java side will be found. Can be a name or an IP address. Can be "localhost". Required if the /pro option is used, and its value is "b" or "h". Ignored otherwise.

/java *javapath*	Java path. *Javapath* is the file path locating the java.exe to be used when autostarting Java. If left out, the first java.exe in the system PATH environment variable will be used.
/javamem *alloc*	Java memory allocation. *alloc* specifies the side of the Java-side memory allocation pool. It is the value that would typically be supplied on a Java command line as part of the –Xmx option. The *alloc* value must be of a format that would be valid if supplied to –Xmx. If the format is invalid, an exception will be thrown. If this option is left out, the Java runtime's default value will be used. If the /nj option is supplied, this value will be ignored.
/jp *jnbcorepath*	JNBCore classpath. *jnbcorepath* is the file path locating the folder containing the file `jnbcore.jar`. If left out, the environment variable CLASSPATH is used to locate `jnbcore.jar`. If using shared-memory communications, this option must be supplied.
/jvm *jvmpath*	JVM. The full path of the jvm.dll file used in shared-memory communications. See *Shared-memory communications* for more information. Required for shared-memory communication, ignored with all other communications mechanisms.
/ls	List all classes to be generated in support of classes (see *Supporting classes*, below), but don't actually generate the proxies. If left out, the proxies will be generated as well as listed.
/n *name*	Name. *name* is the name of the assembly DLL file containing the proxies. If left out, a file `jnbproxy.dll` will be created to contain the proxies.
/nj	No Java. If option is present, the Java-side must be started manually, and the /cp, /jp, and /xp options, if present, are ignored. If left out, the Java-side is started automatically.
/ns	No supporting classes. If option is present, jnbproxy will only generate proxies for the classes specified on the command line, but not for any supporting classes. If left out, proxies for all supporting classes will be generated.
/pd *proxyDirection*	Proxy direction. Specifies the direction in which the proxies will operate. *proxyDirection* may be either **n2j**, in which case it is a .NET-to-Java project and .NET-side proxies in a DLL file will be generated to call Java classes, or **j2n**, in which case it is a Java-to-.NET project and Java-side proxies in a jar file will be generated to call .NET types. If some other proxy direction is supplied, or the /pd option is omitted, the direction will be assumed to be **n2j**.
/port *portNum*	Port. The port on which the Java side will be listening. Must be an integer. Required if the /pro option is used, and its value is "b" or "h". Ignored otherwise.
/pp *propspath*	Property path. *propspath* is the file path for the folder containing the file `jnbcore.properties`. If left out, the value *jnbcorepath* in the /jp option will be used.
/pro *protocol*	Protocol. Specifies the protocol/communication mechanism to be used to communicate between the .NET and Java sides. *protocol* can be either **s** (shared memory), or **b** (binary). If **s**, the /jvm, /bp, and /jp options are required; /nj, /host, and /port are ignored. If **b**, the /host and /port options are required; /jvm is ignored. If the /pro option is omitted, /jvm, /host, and /port are all ignored and the

	.NET-side configuration file jnbproxy.config is used to set up the communications. (Use of jnbproxy.config is supplied for backward compatibility with earlier versions.)
/sn keyFile	Sign. If option is present, the generated assembly will be digitally signed using the strong name key (.snk) file whose full path is given by keyFile. If left out, the assembly will not be signed. No more than one of the /sn and /snd options may be supplied.
/snd keyFile	Sign delayed. If option is present, the generated assembly contain space allowing it to be digitally assigned later according to the .NET delay signing procedure, and will use the public key stored in the strong name key (.snk) file whose full path is given by keyFile. If left out, the assembly will not be set up for delayed signing. No more than one of the /sn and /snd options may be supplied.
/t50	Target Java SE 5.0. If option is present, then Java-side proxies targeted to Java SE 5.0 will be generated. This option must be present if Java-side proxies are to support mappings from .NET generics, enums, and vararg methods. The option is ignored in .NET-to-Java projects.
/vn *versionNum*	Version number. If option is present, the generated assembly will be assigned the supplied version number. *versionNum* must be a valid version number according to the .NET Framework documentation for the class System.Version. If left out, no version number will be assigned to the generated assembly.
/wd *dir*	Working directory. *dir* is the working directory for the Java-side's JVM. This option is ignored if the /nj option is present (that is, if the Java-side is being started manually. If left out, the system's default working directory will be used.

Any other options, specifically those pertaining to Java-to-.NET projects, will be ignored when the /pd option is **n2j** and a .NET-to-Java project is being generated.

Supporting classes

JNBProxy can generate proxies not only for the Java classes that are explicitly listed, but also for *supporting classes*. Informally, a supporting class for a given Java class is any class that might be needed as a direct or indirect result of using that Java class. Formally, for a given Java class, supporting classes include all of the following:

- The given class itself.
- The class's superclass or superinterface (if it exists) and all of its supporting classes.
- The class's implemented interfaces (if any) and all of their supporting classes.
- For each field in the class, the field's class and all of its supporting classes.
- For each method in the class:
  - The method's return value's class (if any) and all of its supporting classes.
  - For each of the method's parameters, the parameter's class and all of its supporting classes.
  - For each exception the method throws, the exception's class and all of its supporting classes.
- For each constructor in the class:

- For each of the constructor's parameters, the parameter's class and all of its supporting classes.
- For each exception the constructor throws, the exception's class and all of its supporting classes.

The number of supporting classes depends on the classes explicitly listed, but will probably be on the order of 200-250 classes. It is recommended that all supporting classes be generated, although there are situations where you, to save time or space, may choose to generate only those classes explicitly specified, without supporting classes.

If a proxy for a supporting class has not been generated, and a proxy for such a class is later needed when the proxies are used, the proxy for the nearest superclass to the required class is used instead. For example, if a proxy for a class C is needed, and the proxy has not been generated, the proxy for the superclass of C will be used if it has been generated. If that proxy hasn't been generated, the proxy for that superclass's superclass will be used if it has been generated, and so forth, until the proxy for `java.lang.Object` (which is always generated) is encountered. Thus, even with an incomplete set of proxies, code will remain functional, although functionality and other information may be lost.

In the GUI version of JNBProxy, **Add** will expose just the explicitly listed classes, whereas **Add +** will additionally expose the supporting classes. In the command line version of JNBProxy, the default is to generate proxies for the supporting classes: use of the /ns option will override this default.

Starting Java manually

JNBProxy uses the Java reflection API to discover information about the 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 which proxies are to be generated.

Both the GUI and command-line versions of JNBProxy can be invoked so that they automatically start and stop the Java-side. In some situations, however, it may be necessary to manually start up the Java-side. For example, the Java-side may be on some other machine, in which case JNBProxy is unable to start it up.

To manually start up the Java-side, the following command-line command must be given:

java –cp *classpath* com.jnbridge.jnbcore.JNBMain /props *propFilePath*

where *classpath* must include:

- The Java classes for which proxies are to be generated (and their supporting classes)
- `jnbcore.jar`

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`.

Alternatively, one can specify one or more properties on the command-line using the /p option:

java –cp *classpath* com.jnbridge.jnbcore.JNBMain /p *name1=value1* /p *name2=value2* …

where each "/p" precedes a name/value properties pair (for example, */p javaSide.serverType=tcp*). The /p option allows individual Java-side properties to be supplied on the command line.

One can also specify both a properties file and individual command-line properties. In such a case, the properties file is read first, then the command-line properties are read, and override any properties of the same name in the properties file.

Note that, if you intend to load an entire jar file using the proxy generation tool, you *must* include the full absolute path of the jar file in the Java classpath, rather than a relative path. For example, if you will be loading x.jar, which is located in C:\A\B, you must include C:\A\B\x.jar in the classpath, even if some other relative path would also be correct. If you do not include the absolute path, the proxy generation tool will report an error.

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

When the Java-side is started manually, an output similar to the following will be seen if the binary protocol is being used:

```
JNBCore v12.0
Copyright 2019, JNBridge LLC

creating binary server
```

If a process (for example, another Java side) is already listening on the new Java side's port, the new Java side will print out an error message and terminate.

Proxy generation example

Assuming that we have the following Java classes, both in the com.jnbridge.samples package:

```
public class Person
{
	public String name;
	public int getID(){…};
	public Person(String name){…};
}

public class Customer extends Person
{
	public int getLastOrder(){…};
	static public int getLastCustomerID(){…};
	public Person getContact(){…};
	public void setContact(Person p){…};
	public void addOrder(String orderInfo) throws InvalidOrderException {…};
	public boolean customerStatus();
	static public void registerCustomer(Customer c){…};
	public Customer(){…};
}

public class InvalidOrderException extends java.lang.Exception
{
}
```

Assuming that CLASSPATH is properly set (so that it includes `jnbcore.jar` and the above classes), issuing the command

```
jnbproxy /pd n2j /pp propspath com.jnbridge.samples.Customer
```

(where *propspath* is the file path for the folder containing the file `jnbcore.properties`)

will result in an assembly `jnbproxies.dll` being created in the current directory containing proxies for com.jnbridge.samples.Customer, com.jnbridge.samples.Person, com.jnbridge.samples.InvalidOrderException, java.lang.Exception, java.lang.String, and all of java.lang.String's and java.lang.Exception's supporting classes.

Proxy generation on .NET Core

To generate .NET proxy DLLs for .NET Core-to-Java projects, you must use a proxy generation tool (either the standalone GUI-based tool, command-line tool, or Visual Studio plugin) from the "traditional" JNBridgePro that works with .NET Framework 4.x. This means that such proxies must be generated on a Windows machine with .NET Framework 4.x installed. The resulting proxies will work with .NET Core applications. The reason that there is no .NET Core-targeted proxy generation tool for .NET Core-to-Java projects is that while .NET Core allows the creation of dynamic assemblies (something that is used in the proxy generation tool) in memory, it does not currently allow these dynamic assemblies to be written out to disk as DLL files. We hope that this will be remedied in future versions of .NET Core (there is a package being developed on GitHub that is designed to do this), but it is not ready for use in a product.

Using proxies with JNBridgePro

System configuration for proxy use

Prior to using the proxies, the system must be properly configured for their use. This section describes the necessary components and configuration details.

.NET-side

Since the .NET Framework supports distributed computing, it is possible that the .NET-side may reside on several machines, where the classes communicate with each other through .NET remoting. Every machine on the .NET-side that communicates with the Java-side must have a local copy of `jnbshare.dll`, and the generated proxy DLL file. Note that every copy of `jnbshare.dll` must be licensed. In addition, each .NET-side must have some method for configuring .NET-side communications. See "Configuring the .NET-side," below.

If shared-memory communication is being used, jnbsharedmem.dll (actually, jnbsharedmem_x86.dll, jnbsharedmem_x64.dll, or both, depending on whether you are creating an x86, x64, or Any CPU project, respectively) must also reside on the machine.

Java-side

There must be a copy of the Java Runtime Environment (JRE), residing on the Java-side. If the Java-side resides on more than one machine, each machine must have a copy of the JRE. The Java-side must also include the Java classes being accessed, and must have a copy of the file `jnbcore.jar` on each machine on which the Java-side resides. The locations of the Java classes and `jnbcore.jar` must either be in the CLASSPATH environment variable, or must be supplied to the Java runtime when it is invoked.

The Java-side properties must somehow be specified. This can be done through the properties file `jnbcore.properties`, through the specification of individual properties on the command-line, or programmatically, through a Properties object supplied as a parameter in a call to JNBMain.start(). See below for more information.

The Java-side for an application may reside on more than one machine, and the .NET classes on one machine may communicate with Java classes on any machines on which a Java-side is installed. To have a .NET-side communicate with multiple Java-sides, see "Interacting with Multiple Java-sides," below.

Configuring the .NET side

The .NET side is configured through the application's configuration file, or programmatically in the user's .NET code. In older versions of JNBridgePro, the .NET-side was configured through use of the special configuration file jnbproxy.config. While use of jnbproxy.config has been deprecated, it is still supported for backward compatibility. Details of jnbproxy.config and its use are given an appendix at the end of the document.

Note that if shared-memory communication is being used, it need only be configured on the client side, not on the server side. So, for example, if you are using shared-memory communication for .NET-to-Java calls, it is only necessary to supply configuration on the .NET side.

Configuring communications through the application configuration file. The application configuration file is the file that is named *app*.exe.config (for an application *app*.exe) or web.config (if it is a configuration file for an ASP.NET Web application).

Inside the <configuration> section of the application configuration file, add the following section if it is not already there:

```
<configSections>
   <sectionGroup name="jnbridge">
   <section name="dotNetToJavaConfig"
              type="System.Configuration.SingleTagSectionHandler, System,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" />
   <section name="javaToDotNetConfig"
              type="System.Configuration.SingleTagSectionHandler, System,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" />
   <section name="tcpNoDelay"
              type="System.Configuration.SingleTagSectionHandler, System,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" />
   <section name="javaSideDeclarations"
              type="System.Configuration.NameValueSectionHandler, System,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" />
   <section name="assemblyList"
              type="com.jnbridge.jnbcore.AssemblyListHandler, JNBShare,
Version=12.10.0.0, Culture=neutral, PublicKeyToken=b18a44fb28aea122" />
   </sectionGroup>
</configSections>
```

Note that for historical reasons, for the 4.8-targeted version of JNBridgePro being used, the "assemblyList" definition above should refer to "JNBShare, Version=12.10.0.0," not "12.0.0.0".

Note: If you are creating an ASP.NET Web application, and the configuration information is in the Web.config file, you will need to fully qualify the configuration section handler classes, as follows:

```
<configSections>
   <sectionGroup name="jnbridge">
   <section name="dotNetToJavaConfig"
```

```
            type="System.Configuration.SingleTagSectionHandler, System,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" />
   <section name="javaToDotNetConfig"
            type="System.Configuration.SingleTagSectionHandler, System,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" />
   <section name="tcpNoDelay"
            type="System.Configuration.SingleTagSectionHandler, System,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" />
   <section name="javaSideDeclarations"
            type="System.Configuration.NameValueSectionHandler, System,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" />
   <section name="assemblyList"
            type="com.jnbridge.jnbcore.AssemblyListHandler, JNBShare,
Version=12.10.0.0, Culture=neutral, PublicKeyToken=b18a44fb28aea122" />
   </sectionGroup>
</configSections>
```

Note that for historical reasons, for the 4.8-targeted version of JNBridgePro being used, the "assemblyList" definition above should refer to "JNBShare, Version=12.10.0.0," not "12.0.0.0".

If there is already a <configSections> section, simply add the <sectionGroup> and <section> tags above to the <configSections> section.

Inside the <configuration> section of the application configuration file, add the following section:

<jnbridge>
… .NET-side specification goes here …
</jnbridge>

To configure a .NET-side client's communications (for .NET-to-Java communication), add one of the following elements inside the <jnbridge> section, depending on whether you are using binary communication (in which case use the first element), or shared-memory communication (in which case use the second element):

```
<dotNetToJavaConfig scheme="jtcp"
                    host="remote host name or IP address"
                    port="remote port number"
                    useSSL="true or false"/>
<!-- if useSSL is true, additional parameters are necessary. See "Secure
Communications using SSL" below -->

<dotNetToJavaConfig scheme="sharedmem"
                    jvm="full path to jvm.dll in JDK or JRE"
                    jnbcore="full path to jnbcore.jar"
                    bcel="full path to bcel-6.n.m.jar"
                    classpath="semicolon-separated Java classpath"/>
```

Note that instead of specifying the *jvm* attribute, you can specify separate 32-bit or 64-bit jvm values to be used as appropriate, depending on whether the application is running as a 32-bit or 64-bit process (for example, if the application has been built as "Any CPU"), as follows:

```
<dotNetToJavaConfig scheme="sharedmem"
                    jvm32="full path to jvm.dll in 32-bit JDK or JRE"
                    jvm64="full path to jvm.dll in 64-bit JDK or JRE"
                    jnbcore="full path to jnbcore.jar"
                    bcel="full path to bcel-6.n.m.jar"
                    classpath="semicolon-separated Java classpath"/>
```

If a value for the *jvm* attribute has been supplied, JNBridgePro will use it. If it has not been supplied, JNBridgePro will use the value for the *jvm32* or *jvm64* attribute, depending on whether this is a 32-bit or 64-bit process. If, at this point, no path to a jvm.dll has been found, an exception will be thrown.

Also note that if a path to a jvm.dll has been found, but it is the wrong bitness (for example, if you are running a 32-bit process and *jvm32* has the path to a 64-bit jvm.dll), an exception will also be thrown.

useSSL is an optional attribute that can be used when configuring tcp/binary (but not shared memory) communications, and indicates whether SSL (secure sockets layer) should be used for secure communications. The value assigned to *useSSL* can be either *true* or *false*. If the *useSSL* attribute is left out, it is equivalent to *useSSL="false"*. See the section "Secure communications using SSL" for more information.

When using shared memory, additional options (for example, the maximum size of the memory allocation pool), they can be supplied through additional `jvmOptions` attributes. For example, to specify that the maximum size of the memory allocation pool should be 80 megabytes, use one of the following configuration elements:

```
<dotNetToJavaConfig scheme="sharedmem"
                    jvm="full path to jvm.dll in JDK or JRE"
                    jnbcore="full path to jnbcore.jar"
                    bcel="full path to bcel-6.n.m.jar"
                    classpath="semicolon-separated Java classpath"
                    jvmOptions.0="-Xmx80m"/>

<dotNetToJavaConfig scheme="sharedmem"
                    jvm32="full path to jvm.dll in 32-bit JDK or JRE"
                    jvm64="full path to jvm.dll in 64-bit JDK or JRE"
                    jnbcore="full path to jnbcore.jar"
                    bcel="full path to bcel-6.n.m.jar"
                    classpath="semicolon-separated Java classpath"
                    jvmOptions.0="-Xmx80m"/>
```

All `jvmOptions` attributes must be of the form `jvmOptions.`*`n`*`="value"` where *n* can be any unique string, but is typically an integer in a sequence starting with 0. Each `jvmOptions` attribute must be associated with exactly one JVM option.

Note: If you use jvmOptions to set the java.library.path property, please be sure to add the path to the version-specific JNBridgePro directory (for example, "4.8-targeted") to the beginning of the path.

A bare-bones application configuration file declaring the standard .NET-side tcp client configuration (communicating via binary communication to a Java server on the same machine listening on port 8085) would look as follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <sectionGroup name="jnbridge">
   <section name="dotNetToJavaConfig"
              type="System.Configuration.SingleTagSectionHandler, System,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" />
   <section name="javaToDotNetConfig"
              type="System.Configuration.SingleTagSectionHandler, System,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" />
   <section name="tcpNoDelay"
              type="System.Configuration.SingleTagSectionHandler, System,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" />
   <section name="javaSideDeclarations"
              type="System.Configuration.NameValueSectionHandler, System,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" />
   <section name="assemblyList"
              type="com.jnbridge.jnbcore.AssemblyListHandler, JNBShare,
Version=12.10.0.0, Culture=neutral, PublicKeyToken=b18a44fb28aea122" />
   </sectionGroup>
   <jnbridge>
```

```
      <dotNetToJavaConfig scheme="jtcp" host="localhost" port="8085" />
 </jnbridge>
</configuration>
```

Note that for historical reasons, for the 4.8-targeted version of JNBridgePro being used, the "assemblyList" definition above should refer to "JNBShare, Version=12.10.0.0," not "12.0.0.0".

Note that either or both .NET-side client and server information may appear in the application configuration file, depending on whether the user is configuring .NET-to-Java calls, Java-to-.NET calls, or both.

If communications configuration appears in the application configuration file, it will override any communication configuration information in jnbproxy.config.

Configuring communications programmatically. .NET-side communications can also be configured programmatically in the user's code. To do so, call one of the overloads of the method

static void com.jnbridge.jnbproxy.JNBRemotingConfiguration.specifyRemotingConfiguration();

There are various ways to call specifyRemotingConfiguration() that are relevant to .NET-to-Java or bidirectional projects:

- specifyRemotingConfiguration(JavaScheme *remoteProtocol*, String *remoteHost*, int *remotePort*,
  bool *useSSL*, string[] *alternateServerNames*,
  string *clientCertificateLocation*, string *clientCertificatePassword*,
  string *clientCertificatePasswordFileLocation*)

  is used to specify a .NET-side client configuration (for .NET-to-Java communication), where *remoteProtocol* is the protocol used to communicate with the Java side, *remoteHost* is the name or IP address of the machine on which the Java side resides (if it is the same machine, it can be "localhost"), and *remotePort* is the port on which the Java side is listening for requests. *useSSL* should be true if SSL (Secure Sockets Layer) should be used for the communications, and should be false otherwise. If *useSSL* is false, the subsequent parameters (*alternateServerNames*, *clientCertificateLocation*, *clientCertificatePassword*, and *clientCertificatePasswordFileLocation*) are ignored. (See "Secure communications using SSL" for more information, including the use of the SSL-specific parameters.) Note that JavaScheme is the enumerated type com.jnbridge.jnbproxy.JavaScheme, and must have the value JavaScheme.binary.

- specifyRemotingConfiguration(JavaScheme *remoteProtocol*, String *remoteHost*, int *remotePort*,
  JavaScheme *localProtocol*, int *localPort*, bool *useSSL*, string *serverCertificateLocation,*
  bool *useIPv6*, string[] *alternateServerNames*,
  string *clientCertificateLocation*, string *clientCertificatePassword*,
  string *clientCertificatePasswordFileLocation*, string *localHostIP*,
  string[] *localIPWhitelist*, bool *useLocalClassWhiteList*,
  string *localClassWhitelistFile*)

  is used to specify both a .NET-side client configuration (for .NET-to-Java communication) and a .NET-side server configuration (for Java-to-.NET communication), where *remoteProtocol* is the protocol with which the .NET side will be sending calls to the Java side, *remoteHost* is the name or IP address of the machine on which the Java side resides (if it is the same machine, it can be "localhost"), *remotePort* is the port on which the Java side is listening for requests, *localProtocol* is the protocol with which the Java side will be sending calls to the .NET side, and *localPort* is the port on which the .NET side will listen for requests from the Java side. *useSSL* should be true if SSL (Secure Sockets Layer) should be used for the communications, and should be false

otherwise. *serverCertificateLocation* is the file path of the X.509 certificate to be used by the server if SSL is to be used. If *useSSL* is false, subsequent SSL-oriented parameters (*serverCertificateLocation*, *alternateServerNames*, *clientCertificateLocation*, *clientCertificatePassword*, *clientCertificatePasswordFileLocation*) are ignored. (See the sections "Secure communications using SSL" in the .NET-to-Java and Java-to-.NET sections of the *Users' Guide* for more information on these parameters and their use.) *useIPv6* should be true if the .NET side should listen to requests on any IPv6 address on the local machine; if it is false, the .NET side should listen for requests to any IPv4 address on the local machine. *localHostIP* specifies the IP address to which the .NET side should be bound when it responds to requests from a Java side. (See "Binding an IP address.") *localIPWhitelist* is the list of IP addresses of Java clients that are allowed to make calls to the .NET side. (See "Whitelisting Java clients.") *useLocalClassWhiteList* specifies whether class whitelisting should be used on the .NET side in Java-to-.NET calls, and (if *useLocalClassWhiteList* is true) *localClassWhiteListFile* is the path to the file containing the class whitelist. (See "Class whitelisting.") Note that JavaScheme is the enumerated type com.jnbridge.jnbproxy.JavaScheme, and must have the value JavaScheme.binary.

Note: Use of IPv6 is only supported when running on operating systems that support IPv6. If the underlying operating system does not support IPv6 and *useIPv6* is set to true, an exception will be thrown.

- specifyRemotingConfiguration(JavaScheme *remoteProtocol*, String *jvmLocation*,
  String *jnbcoreLocation*, String *bcelLocation*, String *classpath*, String[] *jvmOptions*)

  is used to specify a .NET-side client configuration that uses shared-memory communication to communicate between the .NET and Java sides. *remoteProtocol* must be JavaScheme.sharedmem, or an exception will be thrown. *jvmLocation* is the full path to the file jvm.dll which implements the Java Virtual Machine and which is typically distributed with a JDK (Java Development Kit) or JRE (Java Runtime Environment). *jnbcoreLocation* is the full path to jnbcore.jar. *bcelLocation* is the full path to bcel-*6.n.m*.jar. *classpath* is the semicolon-separated classpath to be used by the Java side. *jvmOptions* is a string array containing any additional options to be supplied to the JVM upon startup. For example, if you wish to specify that the maximum side of the JVM's memory allocation pool should be 80 megabytes, *jvmOptions* should contain the string "-Xmx80m".

- specifyRemotingConfiguration(JavaScheme *remoteProtocol*, String *jvmLocation*,
  String *jnbcoreLocation*, String *bcelLocation*, String *classpath*)

  is a convenience version of the previous version of specifyRemotingConfiguration(), used when shared-memory communication is to be used and there are no additional JVM options to be supplied. It is equivalent to calling

  specifyRemotingConfiguration(*remoteProtocol*, *jvmLocation*, *jnbcoreLocation*,
  bcelLocation, *classpath*, new String[0]);

- specifyRemotingConfiguration(JavaScheme *remoteProtocol*, String *jvmLocation*,
  String *jnbcoreLocation*, String *bcelLocation*, String *classpath*, String[] *jvmOptions*,
  String *javaEntryLocation*)

  is used to configure bi-directional shared-memory communication when the program is started on

the .NET side. See the section "Bi-directional shared-memory communications," below, for more information.

- specifyRemotingConfiguration(JavaScheme *remoteProtocol*, String *jvmLocation*,
  String *jnbcoreLocation*, String *bcelLocation*, String *classpath*, String *javaEntryLocation*)

  is a convenience version of the previous version of specifyRemotingConfiguration(), used when bi-directional shared-memory communication (started from the .NET side) is to be used and there are no additional JVM options to be supplied. It is equivalent to calling

  specifyRemotingConfiguration(*remoteProtocol*, *jvmLocation*, *jnbcoreLocation*,
  bcelLocation, *classpath*, new String[0], *javaEntryLocation*);

  See the section "Bi-directional shared-memory communications," below, for more information.

A call to specifyRemotingConfiguration() must be made before any access of a proxy object in order for it to have an effect. If the call to specifyRemotingConfiguration() is made after an access of a proxy object (for example, a call to a constructor or an access of one of its members), a RemotingException will be thrown.

If a call to specifyRemotingConfiguration() is made, it will override any communications configuration information in the application configuration file or in jnbproxy.config.

Note that a number of "convenience APIs" available up through JNBridgePro 10.1, that left out various parameters, have been removed from JNBridgePro. Now, when using TCP/binary communications, all parameter values must be explicitly given. For configuration of shared memory, we are still providing convenience APIs.

Issues in configuring the .NET side when using shared-memory communications

When using shared-memory communications, you must make sure that the account running the program has "Read & Execute," "List Folder Contents," and "Read" access permissions to the folders containing the Java class and jar files used by the programs (including jnbcore.jar and bcel-*6.n.m*.jar). If these permissions are not present, a ClassNotFoundException or a NoClassDefFoundError may be thrown. It is particularly necessary to do this when creating ASP.NET Web applications using shared memory, since the ASPNET account that typically runs ASP.NET Web apps may not have access to these class and jar files.

It is also sometimes the case that the ClassNotFoundException or NoClassDefFoundError can be thrown because of a known issue involving interactions between shared memory and Java classloaders. To address this problem, it is necessary to move one or more classes or jar files from the regular classpath to the *boot classpath*. The actual procedure for doing this depends on how you are configuring JNBridgePro.

If you are configuring JNBridgePro using the application configuration file, you will have an element in the application configuration file like:

```
<dotNetToJavaConfig
        scheme="sharedmem"
        jvm="full path to jvm.dll in JDK or JRE"
        jnbcore="full path to jnbcore.jar"
        bcel="full path to bcel-6.n.m.jar"
        classpath="semicolon-separated Java classpath"/>
```

Change this element so it looks like:

```
<dotNetToJavaConfig
        scheme="sharedmem"
        jvm="full path to jvm.dll in JDK or JRE"
        jnbcore="full path to jnbcore.jar"
        bcel="full path to bcel-6.n.m.jar"
        classpath="semicolon-separated Java classpath"
        jvmOptions.0="-Xbootclasspath/p:semicolon-separated boot classpath goes here"/>
```

Remove the jar file in question from the classpath above, and put it in the boot classpath.

If you are configuring JNBridgePro using jnbproxy.config, you do something similar. In jnbproxy.config, change

```
<channel type="com.jnbridge.jnbproxy.JNBSharedMemChannel, JNBShare, Version=12.10.0.0,
Culture=neutral, PublicKeyToken=b18a44fb28aea122"
        jvm="location of jvm.dll distributed with JDK or JRE"
        jnbcore="location of jnbcore.jar"
        bcel="location of bcel-6.n.m.jar"
        classpath="semicolon-separated classpath"/>
```

to

```
<channel type="com.jnbridge.jnbproxy.JNBSharedMemChannel, JNBShare, Version=12.10.0.0,
Culture=neutral, PublicKeyToken=b18a44fb28aea122"
        jvm="location of jvm.dll distributed with JDK or JRE"
        jnbcore="location of jnbcore.jar"
        bcel="location of bcel-6.n.m.jar"
        classpath="semicolon-separated classpath"
        jvmOptions.0="-Xbootclasspath/p:semicolon-separated boot classpath goes here"/>
```

and move classes and jar files from the classpath to the boot classpath as described above.

Note that for historical reasons, for the 4.8-targeted version of JNBridgePro being used, the <channel> elements above should refer to "JNBShare, Version=12.10.0.0," not "12.0.0.0".

If you're configuring JNBridgePro programmatically, use

```
JNBRemotingConfiguration.specifyRemotingConfiguration(JavaScheme remoteProtocol, String
jvmLocation, String jnbcoreLocation, String bcelLocation, String classpath, String[] jvmOptions)
```

where jvmOptions is a string array containing one element with the string
"-Xbootclasspath/p:*semicolon-separated boot classpath goes here*"

and adjust the classpath and the boot classpath similarly.

If you see a new ClassNotFoundException, or NoClassDefFoundError, repeat the process by moving jar files from the classpath to the boot classpath until the problem disappears.

.NET-side configuration in .NET Core

One of the biggest differences between traditional JNBridgePro and JNBridgePro for .NET Core is that the .NET Core version uses JSON to specify JNBridgePro configuration. This is because .NET Core applications do not use app.config files. If configured using configuration files rather than programmatically, an application using JNBridgePro for .NET Core must have a file jnbridgeConfig.json, with the following form:


```
{
  "dotNetToJavaConfig": {
    "scheme": "jtcp or sharedmem",
    "host": "remote host name or IP address",
    "port": remotePortNumber,
    "useSSL": "true or false",
    "SSLAlternateServerNames":  [ "list", "of", "acceptable server names" ],
    "SSLClientCertificateLocation": "path to client certificate",
    "SSLClientCertificatePassword": "client certificate password, if necessary",
    "SSLClientCertificatePasswordFileLocation":
       "path to file containing client certificate password, if necessary",
    "jvmOptions": [ "list", "of", "jvm options" ],
    "jvm": "path to jvm.dll if using shared memory",
    "jvm32": "path to 32-bit jvm.dll if using shared memory",
    "jvm64": "path to 64-bit jvm.dll if using shared memory",
    "jnbcore": "path to jnbcore.jar",
    "bcel": "path to bcel-6.n.m.jar",
    "classpath": "semicolon-separated classpath"
  },
  "javaToDotNetConfig": {
    "scheme": "jtcp",
    "port": "portNumber",
    "useSSL": "true or false",
    "certificateLocation": "path to SSL certificate",
    "useIPv6": "true or false"
    "hostIP": "IP address for which .NET side will take requests",
    "ipWhitelist": [ "list", "of", "whitelisted client addresses"],
  },
  "assemblyList": [ "list", "of", "assembly paths"],
  "tcpNoDelay": "true or false",
  "licenseLocation": {
    "host": "host",
    "port": "portNumber",
    "directory": "C:\\Program Files (x86)\\JNBridge\\JNBridgePro v12.0"
  },
  "javaSideDeclarations": [
    {
      "javaSideName": "second_PrimaryURL",
      "javaSideURL": "jtcp://localhost:8090/JNBDispatcher"
    },
    {
      "javaSideName": "js2",
      "javaSideURL": "url2"
    }
  ]
}
```

The elements above correspond directly to elements in the traditional JNBridgePro app.config file. Clearly, not all elements are necessary, and some sets of elements are mutually exclusive. See the appropriate sections of the *JNBridgePro Users' Guide* for more information on how these elements are used.

The file jnbridgeConfig.json must be in the same folder as the copy of jnbshare.dll being used by your application.

Note that, while the JSON specification does not allow for comments (a serious oversight, in our opinion), Microsoft's implementation does support comments. Everything from a '//' sequence to the end of a line is considered a comment and ignored.

Programmatic configurating using the JNBRemotingConfiguration.specifyRemotingConfiguration() APIs is also available and works the same way as it does in "traditional" JNBridgePro.

Example: .NET Core-to-Java projects using TCP/binary communications

```
{
  "dotNetToJavaConfig": {
    "scheme": "jtcp",
    "host": "remote host name or IP address",
    "port": remotePortNumber,
    "useSSL": "true or false",     // optional.
                                   // If true, additional properties are
                                   // necessary
  },
  "tcpNoDelay": "true or false",   // optional
  "licenseLocation": {      // use host/port, or directory, but not both
    "host": "host",
    "port": "portNumber",
    "directory": "C:\\Program Files (x86)\\JNBridge\\JNBridgePro v12.0"
  },
  "javaSideDeclarations": [ // optional: only use with multiple Java sides
    {
      "javaSideName": "second_PrimaryURL",
      "javaSideURL": "jtcp://localhost:8090/JNBDispatcher"
    },
    {
      "javaSideName": "js2",
      "javaSideURL": "url2"
    }
  ]
}
```

Example .NET Core-to-Java projects using shared memory communications (Windows)

```
{
  "dotNetToJavaConfig": {
    "scheme": "sharedmem",
    "jvmOptions": [ "list", "of", "jvm options" ],     // optional
      // if jvm32 and jvm64 are both used, the one of the
      // appropriate process bitness will be used.
      // jvm is available for backward compatibility, but is deprecated
    "jvm": "path to jvm.dll ",
    "jvm32": "path to 32-bit jvm.dll",
    "jvm64": "path to 64-bit jvm.dll ",
    "jnbcore": "path to jnbcore.jar",
    "bcel": "path to bcel-6.n.m.jar",
    "classpath": "semicolon-separated classpath"
  },
  "licenseLocation": {      // use host/port, or directory, but not both
```

```
        "host": "host",
        "port": "portNumber",
        "directory": "C:\\Program Files (x86)\\JNBridge\\JNBridgePro v12.0"
      }
    }
```

Example .NET Core-to-Java projects using shared memory communications (Linux)

```
{
  "dotNetToJavaConfig": {
    "scheme": "sharedmem",
    "jvm64": "/usr/lib/jvm/java-8-openjdk-amd64/jre/lib/amd64/server/libjvm.so",
    "jnbcore": "/home/myName/N2JProject/netcoreapp3.0/jnbcore.jar",
    "bcel": "/home/myName/N2JProject/netcoreapp3.0/bcel-6.n.m.jar",
    "classpath":
"/home/myName/N2JProject/netcoreapp3.0/Java:/home/citrin/Downloads/netcoreapp3.0/Java
2nd Proxy"
  },
  "licenseLocation": {
    "directory": "/home/myName/N2JProject/netcoreapp3.0"
  }
}
```

Java-side properties and the jnbcore.properties file

You can change various configuration properties of the Java side by editing the jnbcore.properties file and re-starting the Java side. Alternatively, you can supply these properties individually on the command-line, or in a Properties table object when calling JNBMain.start(). The jnbcore.properties file contains the following properties related to .NET-to-Java interoperability that can be altered:

- **javaSide.serverType**: This property indicates the communications protocol to be used. The associated value must be tcp (for tcp/binary communications). Any other value will cause an exception to be thrown. This property must be defined; if it is not, an exception will be thrown.
- **javaSide.port**: This property indicates the port on which the Java side will be listening for requests. It must be an integer, and must be a legal port value. If the port is already being used by another process, an error message will be displayed and the Java side will shut down. If this property is missing, the default port value, 8085, will be used.
- **javaSide.timeout**: This property indicates the number of milliseconds before a listener socket times out. If the property is missing, a default value of 5000 milliseconds will be used. You should not have to change this value.
- **javaSide.workers**: This property indicates the number of listener threads to be created in the thread pool when the Java side starts. If more threads are needed, they will be created automatically. If this property is missing, a default value of 5 will be used. Increasing this value may provide a performance improvement if the Java side will be receiving requests from a large number of connections.
- **javaSide.backlog**: This property indicates the number of backlog requests from the .NET side that can be waiting to be serviced by the Java side before the Java side begins to refuse requests. If the property is missing, the default value of 50 will be used. Increasing this value may be necessary if the Java side will be receiving a large number of requests very frequently from a large number of connections.

- **javaSide.hostIP**: This property indicates the host name or IP address on which the Java side will be listening for requests. It may be a host name, or a valid IP address. If this property is missing, the Java side will respond to requests sent to any and all local addresses. This property is generally used when a host has several names or IP addresses, and the Java side should only respond to requests sent to one of those.

  Note: When javaSide.hostIP is left out, the Java side will respond to requests on any and all local *IPv4* addresses. To have the Java side respond to any and all *IPv6* addresses, supply the IPv6 wildcard value *::* (two colons) for the javaSide.hostIP property. Similarly, to have the Java side respond to a particular IPv6 address, supply the address in IPv6 hex-colon format, or supply a host name that resolves to an IPv6 address.

  Note: Use of IPv6 is only supported when running on operating systems that support IPv6. If the underlying operating system does not support IPv6 and an IPv6 address is used, an exception will be thrown.

- **javaSide.useSSL**: This property indicates whether the Java-side server should use SSL (secure sockets layer) to support secure communications. The value can be *true* or *false*. If the property is missing, or if it has some other value, the default value of *false* will be used. If javaSide.useSSL is true, you must also specify the properties **javaSide.keyStore**, **javaSide.keyStorePassword**, **javaSide.trustStore**, and **javaSide.trustStorePassword**. See "Secure communications using SSL," for more information.

- **javaSide.noCompress**: This property indicates whether data compression should be used. The value can be *true* or *false*. If the property is missing, or if it has some other value, the default value of *false* will be assumed. When the property is true, the Java side will never send a message data compression will never be used. Otherwise, it will be used when large messages are sent between the .NET and Java sides. The Java side will *understand* compressed messages when it receives them, even when this property is true. Note that this property is only used for tcp/binary communications. When other communications methods are used, this property will be ignored.

Configuring and starting a standalone JVM for proxy use

Note: If your .NET-side client application has been configured to use shared-memory communication, you do not need to explicitly start a standalone JVM; it will be run automatically. The instructions below only apply if you are using binary communication.

Applications using JNBridgePro to access Java classes and objects from .NET must have an installed and running Java side. This can be either a standalone JVM, or a Java EE application server. Information on configuring and running a Java EE application server to work with JNBridgePro is given elsewhere in the *Users' Guide*. If a standalone JVM is to be used, a JVM must be running that contains the JNBCore component and the Java classes for which proxies are to be generated.

To manually start up the Java-side in a standalone JVM, the following command-line command must be given:

java –cp *classpath* com.jnbridge.jnbcore.JNBMain /props *propFilePath*

where *classpath* must include:

- The Java classes for which proxies are to be generated (and their supporting classes)
- `jnbcore.jar`

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`.

Alternatively, one can specify one or more properties on the command-line using the /p option:

java –cp *classpath* com.jnbridge.jnbcore.JNBMain /p *name1=value1* /p *name2=value2* …

where each "/p" precedes a name/value properties pair (for example, */p javaSide.serverType=tcp*). The /p option allows individual Java-side properties to be supplied on the command line.

One can also specify both a properties file and individual command-line properties. In such a case, the properties file is read first, then the command-line properties are read, and override any properties of the same name in the properties file.

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

When the Java-side is started manually, an output similar to the following will be seen if the binary protocol is being used:

```
JNBCore v12.0
Copyright 2019, JNBridge, LLC

creating binary server
```

Configuring and starting a Java side programmatically from within an existing JVM

Note: The following instructions only apply if you are using binary communication. Shared-memory communication cannot be used to communicate with an existing JVM.

To start a Java side programmatically from within an existing JVM running a Java program, call

com.jnbridge.jnbcore.JNBMain.start(String *propertiesFilePath*)

where *propertiesFilePath* is the full path of the jnbcore.properties file containing the javaSide.* properties, or call

com.jnbridge.jnbcore.JNBMain.start(java.util.Properties *properties*)

where *properties* is a properties table containing the javaSide.* properties. Use of the second form of the call allows Java-side properties to be computed programmatically.

Once a Java side is running, it can be stopped by calling

com.jnbridge.jnbcore.JNBMain.stop()

The class JNBMain is in jnbcore.jar, and jnbcore.jar must therefore be in the classpath of any program calling JNBMain.start() and JNBMain.stop() during compilation and execution.

Secure communications using SSL (Secure Sockets Layer)

SSL is a security feature in JNBridgePro. For a unified discussion of security in JNBridgePro, see the section "JNBridgePro and security."

JNBridgePro supports secure cross-platform communications using SSL (secure sockets layer). SSL is turned on by default when using TCP/binary communications, and includes server authentication,

client authentication, and encryption. Server authentication ensures that your .NET sides are communicating with a particular Java side, and prevents other servers from pretending to be your Java side. Client authentication ensures that only specifically permitted .NET sides can access the Java side, but no others. Encryption ensures that nobody else can eavesdrop on the TCP/binary communications between the .NET and Java sides.

SSL is turned off by default, since configuring it is a bit more complex than the other security features. You can leave it off, but you will be opening yourself to a number of potential attacks, including the ability of any client to execute any code on your Java side's machine. (IP and class whitelisting can mitigate these hazards.)

SSL is not used in shared memory communication. With shared memory, the .NET and Java sides run in the same process, and communication is done using shared data structures, and is inherently secure.

To implement SSL, you need the following certificates for each .NET and Java side.

Each Java side requires an X.509 certificate, typically contained in a .cer file. In the context of .NET-to-Java interop, this will be known as a *server certificate*. The *common name* (CN) field of the server certificate should be the name of the server on which the Java side resides.

Each .NET side requires two files: an X.509 certificate, typically contained in a .cer file, and a PKCS#12 or PFX file containing the public key (also in the corresponding .cer file) and the private key, and is typically contained in a .p12 or .pfx file. The .p12 and .pfx files typically have an associated password. In the context of .NET-to-Java communications, these .cer and .p12/.pfx files are known as *client certificates*.

On the Java-side machine, install the server certificate in a Java *keystore* (.jks) file. Note that there are many tools to create and manage certificates and .jks files, including the keytool that comes with the JDK, and the free, open source, GUI-based KeyStore Explorer. All should work, although we find the KeyStore Explorer particularly easy to use.

Also, on the Java-side machine, install each authorized .NET side's client certificate (the .cer file, not the corresponding .p12/.pfx file) in a different .jks file known as the *truststore*. Note that the keystore and the truststore should have passwords associated with each.

On each .NET-side machine, install the .p12/.pfx file in that machine's certificate store. Do so by double-clicking on the file in Windows Explorer or by right-clicking and selecting Install certificate…, and following the instructions. *(If the certificate is self-signed, it must be imported into either the* ***Trusted Root Certification Authorities*** or ***Third-Party Root Certification Authorities*** *stores. When the installation asks you where the certificate should be stored, you must select one of those options.)* Also, leave the client .cer certificate file on .NET-side's disk drive – you will be supplying the path to the .cer file as part of the configuration. After installing the .p12/.pfx, you can remove it from the .NET-side machine's file system, although you should keep a copy in a secure place.

Configure the Java side as follows by supplying the following properties, either in the Java-side properties file, or programmatically:

- javaSide.useSSL=true # if this is left out, the value is false by default
- javaSide.keyStore=*path_to_keystore_.jks_file* # the path to the file itself, not to the folder
- javaSide.keyStorePassword=*keystore_password*
- javaSide.trustStore=*path_to_truststore_.jks_file*

- javaSide.trustStorePassword=*truststore_password*

Note that the keystore and truststore passwords should probably be configured programmatically from the contents of a secure file, or from values input by users. Placing them inside the .properties file, where they can be read by anyone, is probably a bad idea.

On the .NET side, if you are configuring the .NET side using an app.config or web.config file, use the following <dotNetToJavaConfig> element:

```
<dotNetToJavaConfig
    scheme="jtcp"
    host="hostname"
    port="port number"
    useSSL="true"
    sslAlternateServerNames="semicolon-separated list of server names"
    clientCertificateLocation="path to .cer/.p12/.pfx file"
    clientCertificatePassword="the certificate's password"
    clientCertificatePasswordFileLocation="path to a file containing the certificate's password"
/>
```

If sslAlternateServerNames is not supplied, then the only Java-side server certificate that will be accepted must have a Common Name (CN) field identical to the value supplied in the host="*hostname*" element. For example if host="localhost", then if the CN of the server certificate is anything but "localhost," an exception will be thrown. If the CN is "myServer" or "myServer.myDomain.com", etc., and the .NET side was configured to use host="*hostname*", an exception will be thrown. To accommodate this, supply a semicolon-separated list of server names that will also be accepted. For example, in the above scenario, sslAlternateServerNames="myServer;myServer.myDomain.com" will avoid an exception.

clientServerLocation is assigned the path to the client certificate – the .cer, .p12, or .pfx file. It must be a path to the file itself, not to the containing folder. If the certificate does not have a password (as most .cer files do not), clientCertificatePassword and clientCertificatePasswordFileLocation will be ignored. If the certificate has a password, it will be the one supplied by clientCertificatePassword, or the one in the first line of the text file pointed to by clientCertificatePasswordFileLocation. If clientCertificatePassword and clientCertificatePasswordFileLocation are both supplied, only clientCertificatePasswordFileLocation will be used; clientCertificatePassword will be ignored. We do not recommend using clientCertificatePassword, as it is insecure; use clientCertificatePasswordFileLocation and store the password in a file that can only be read by the .NET side application.

SSL can also be configured programmatically on the .NET side. For more information, please see the section on the programmatic configuration APIs.

SSL can be turned off. On the Java side, set the property javaSide.useSSL=false. On the .NET side, use the app.config element

```
<dotNetToJavaConfig
    scheme="jtcp"
    host="hostname"
    port="port number"
    useSSL="false"
/>
```

If SSL is turned off, it must be turned off in all .NET sides and Java sides that participate in the application. Again, note that if SSL is turned off, you open yourself to other attacks, and should mitigate them by employing other security mechanisms offered by JNBridgePro (including IP and class whitelisting), making sure that your firewall is properly configured, or using shared memory instead.

Class whitelisting

Class whitelisting is a security feature in JNBridgePro. For a unified discussion of security in JNBridgePro, see the section "JNBridgePro and security."

Starting with JNBridgePro 10.1, it is possible to specify which classes' APIs are available for cross-platform access, through the class whitelisting feature. Without whitelisting, it is possible for unauthorized clients to access APIs on the Java-side machine. It is even possible for them to execute arbitrary code on the Java side machine through use of the Runtime.exec() API, and it is even possible for this to happen from clients that are not using the JNBridgePro proxies and runtime components. Client whitelisting mitigates this problem by only allowing specific APIs to be accessed remotely.

When class whitelisting is turned on (which it is by default), a list of classes is read from a text file known as the class whitelist file. The file contains one class per line, and should contain no additional commas or other delimiters. (Whitespace surrounding the class names is allowed.) The following is an example of the contents of a class whitelist file:

```
pkg1.Class1
pkg2.Class2
pkg3.Class3
```

If a class is whitelisted, then the Java side will allow calls to instance members of objects of that class, and it will allow calls to static members and constructors of that class.

In addition, if an interface is whitelisted, then any other class that implements the interface is considered whitelisted. Similarly, if a class is whitelisted, then any other class that inherits from the whitelisted class is also considered whitelisted. Finally, if a class is whitelisted, then all of its nested classes are considered whitelisted.

The only exception to the above is java.lang.Object. It is whitelisted by default, so that methods declared in the Object class are accessible, but classes that inherit from Object (that is, all classes) are not automatically whitelisted – otherwise, all classes would be whitelisted.

In addition to Object, other classes that are automatically whitelisted are Class, String, Integer, Short, Long, Float, Double, Boolean, Byte, and Char.

All other classes must be explicitly whitelisted, or inherit from or implement classes or interfaces that have been explicitly whitelisted.

If a class has not been whitelisted, then any access to that class, or to objects of that class, from a .NET client or any other client over a TCP/binary connection, will cause an exception to be thrown.

The class whitelist is specified through the following Java-side property:

```
javaSide.classWhiteListFile=path to white list file
```

For example,

```
javaSide.classWhiteListFile=C:/Documents/classWhiteList.txt
```

It is also possible to turn off class whitelisting through the following Java-side property:

```
javaSide.useClassWhiteList=false
```

We only recommend turning off whitelisting during the early stages of development. During testing and deployment, we strongly recommend using whitelisting.

Whitelisting is only offered when TCP/binary communications is used. It is not offered when shared memory is used, since remote clients will not be accessing the Java side, and it is therefore not necessary.

Whitelisting .NET clients

Client IP whitelisting is a security feature in JNBridgePro. For a unified discussion of security in JNBridgePro, see the section "JNBridgePro and security."

By default, for security reasons, a JNBridgePro Java side will only accept requests from .NET sides. It may be desirable to specify a whitelist of hosts from which a Java side will accept .NET requests. To specify such a whitelist, add the following property to the Java-side specification:

dotNetSide.ipWhitelist=*semicolon-separated list of IP addresses*

The IP addresses may be in IPv4 ('.'-separated) or IPv6 (":"-separated) style. "*" may be used as a wildcard in place of any element. "*" or "::" may be used as a wildcard to match any IP address.

If the dotNetSide.ipWhilelist property is not specified, it is equivalent to having specified

dotNetSide.ipWhitelist=127.0.0.1;::1

that is, the Java side will only accept requests from clients on the same machine.

As an example,

dotNetSide.ipWhitelist=1.2.3.4

will cause the Java side to only accept requests from the machine whose IP address is 1.2.3.4. All other requests will be rejected, and any .NET side making a proxy call to that Java side from any other machine will see a System.IO.IOException.

As another example,

dotNetSide.ipWhitelist=127.0.0.1;10.1.2.*

will cause the Java side to only accept .NET requests from the same host, or from any host in the range 10.1.2.0 to 10.1.2.255, and reject all others.

Interacting with multiple Java-sides (Server license only)

It is possible to have a .NET-side on one machine communicate with multiple Java-sides. These Java-sides may be on multiple machines, or there may be several JVMs on one machine, each running its own Java-side. A .NET-side may use a different communication protocol in communication with each Java-side; there is no requirement that the same protocol be used in all cases.

Each Java-side is configured and started in the usual manner. If multiple Java-sides are running on one machine, then each Java-side must be configured to listen on a different port, which implies that each Java-side must have its own properties file.

A .NET-side always has exactly one *default Java-side* with which it communicates. The location (host name and port) of the default Java-side, and the protocol used to communicate with it, are contained in the application configuration file, or specified programmatically through a call to

JNBRemotingConfig.specifyRemotingConfiguration(), either of which must be present. Additional Java-sides can be declared in the .NET program running on the .NET-side. At any given time, there is exactly one *active* Java-side; when the program starts, the default Java-side is always the active Java-side, but the active Java-side can be changed at any time to be one of the other declared Java-sides or back to the default Java-side.

When a proxy is instantiated, that instance will always communicate with whatever Java-side was active at the time of its creation, even if the active Java-side is subsequently changed. Thus, calls to any instance methods or fields in that proxy will always go to the Java-side that was active at the time of the proxy creation.

In contrast, all calls to static methods and fields are sent to the Java-side that is active at the time of the call.

In all cases, it is the responsibility of the user to make sure that the desired Java class is present on the target Java-side. If it is not, a ClassNotFoundException or NoClassDefFoundError will be thrown.

Also, it is currently not permissible to pass a reference to a Java object on one Java-side as a parameter to a Java method residing on a different Java-side. The results of such an operation are undefined: an exception might or might not be thrown, and the results will most likely be incorrect. It is the responsibility of the user to make sure that this does not happen. This limitation will be removed in an upcoming version.

The JNBridgePro class com.jnbridge.jnbproxy.JavaSides provides an API for declaring new Java-sides and for changing the active Java-side. (It is also possible to declare new Java-sides in the application configuration file. This will be described at the end of this section.)

- **static void addJavaServer(string name, JavaScheme scheme, string host, int port)**
  Declare a new Java-side with which the current .NET side can communicate. *name* is the name of the new Java-side and must not have been previously used, or a RemotingException will be thrown. *scheme* is a value of the enumerated type com.jnbridge.jnbproxy.JavaScheme, and must be **binary**. *host* is the name of the host on which the Java-side resides, and can be either a resolvable host name, "localhost" if the Java-side is on the same machine, or an IP address. *port* is the port on which the Java-side is listening for communications.
  Note: the name "default" is reserved for the default Java-side, and may not be used as a value for the name argument in addJavaServer().

- **static void setJavaServer(string name)**
  Set the named Java-side to be the new current Java-side. A Java-side of the specified name must have been previously declared using addJavaServer(), or a RemotingException will be thrown.

- **static void resetDefaultJavaServer()**
  Set the default Java-side to be the new current Java-side.

- **static string[] servers()**
  Return the names of all active servers.

As an example, assume that the default Java-side is on the same host as the .NET-side (localhost), listens on port 8085, and communicates via the binary protocol. The following example code creates a second Java-side on host "other," listing on port 8090, and using binary communications, then creates instances of a class C on each of the Java-sides, then finally calls static class methods of C on each Java-side:

```
JavaSides.addJavaServer("otherJSide", JavaScheme.binary, "other", 8090);
```

```
C c1 = new C();  // created on the default Java-side
JavaSides.setJavaServer( "otherJSide" );    // otherJSide is now active
C c2 = new C();  // created on otherJSide
c1.instanceMethod();    // sent to default Java-side
c2.instanceMethod();    // sent to otherJSide
C.staticMethod();       // sent to otherJSide (currently active)
JavaSides.resetDefaultJavaServer();  // default is now active
C.staticMethod();       // sent to default Java-side
```

Note that all the parameters in the above JavaSides API calls can be variables, and that the active Java-side can therefore be determined programmatically.

Declaring additional Java-sides in the application configuration file

It is possible to declare additional Java sides in the application configuration file. This is the file that is named app.exe.config (for an application app.exe) or web.config (if it is a configuration file for an ASP.NET Web application).

Inside the <configuration> section of the application configuration file, add the following section:

```
<configSections>
   <sectionGroup name="jnbridge">
   <section name="dotNetToJavaConfig"
              type="System.Configuration.SingleTagSectionHandler" />
   <section name="javaToDotNetConfig"
              type="System.Configuration.SingleTagSectionHandler" />
   <section name="tcpNoDelay"
              type="System.Configuration.SingleTagSectionHandler" />
   <section name="javaSideDeclarations"
              type="System.Configuration.NameValueSectionHandler" />
   <section name="assemblyList"
              type="com.jnbridge.jnbcore.AssemblyListHandler, JNBShare,
Version=12.10.0.0, Culture=neutral, PublicKeyToken=b18a44fb28aea122" />
   </sectionGroup>
</configSections>
```

Note that for historical reasons, for the 4.8-targeted version of JNBridgePro being used, the "assemblyList" definition above should refer to "JNBShare, Version=12.10.0.0," not "12.0.0.0".

If there is already a <configSections> section, simply add the <sectionGroup> and <section> tags above to the <configSections> section.

Inside the <configuration> section of the application configuration file, add the following section:

```
<jnbridge>
  <javaSideDeclarations>
  … new Java-side declarations go here …
  </javaSideDeclarations>
</jnbridge>
```

A new Java-side declaration for a Java-side named *X* on host *Host* and port *Port*, using protocol *Protocol* (jtcp, if binary/tcp) is as follows:

```
<add key="X_PrimaryURL" value="Protocol://Host:Port/JNBDispatcher" />
```

For example, in the programmatic example above, instead of using a call to
`JavaSides.addJavaServer("otherJSide", JavaScheme.binary, "other", 8090);`
one could place the following declaration in the application configuration file:

```
<add key="otherJSide_PrimaryURL" value="jtcp://other:8090/JNBDispatcher" />
```

A bare-bones application configuration file declaring otherJSide would look as follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <configSections>
   <sectionGroup name="jnbridge">
   <section name="dotNetToJavaConfig"
            type="System.Configuration.SingleTagSectionHandler" />
   <section name="javaToDotNetConfig"
            type="System.Configuration.SingleTagSectionHandler" />
   <section name="tcpNoDelay"
            type="System.Configuration.SingleTagSectionHandler" />
   <section name="javaSideDeclarations"
            type="System.Configuration.NameValueSectionHandler" />
   <section name="assemblyList"
            type="com.jnbridge.jnbcore.AssemblyListHandler, JNBShare,
Version=12.10.0.0, Culture=neutral, PublicKeyToken=b18a44fb28aea122" />
   </sectionGroup>
   </configSections>
   <jnbridge>
      <javaSideDeclarations>
      <add key="otherJSide_PrimaryURL" value="jtcp://other:8090/JNBDispatcher" />
      </javaSideDeclarations>
   </jnbridge>
</configuration>
```

Note that for historical reasons, for the 4.8-targeted version of JNBridgePro being used, the "assemblyList" definition above should refer to "JNBShare, Version=12.10.0.0," not "12.0.0.0".

Note that you should not declare the same additional Java-side both programmatically (using addJavaServer()) and declaratively (using the application configuration file). If you do, an exception will be thrown.

Also note that you must declare the default Java-side somewhere. This can be in the application configuration file (using the <dotNetToJavaConfig> tag), or programmatically from the user code. See the section "Configuring the .NET side," elsewhere in this document.

Note on generating proxies on multiple Java-sides: It is only possible to have the jnbproxy tool communicate with one Java-side at a time. If all the Java-sides offer the same set of classes, a single proxy assembly can be used to communicate with all Java-sides if the JavaSides API is used as above. If different Java-sides offer different classes, one may create proxy assemblies for each set using a separate run of the jnbproxy tool. The user is still responsible for indicating which Java-side is to be used through the JavaSides API. Also, if multiple proxy assemblies have been generated, the same class may be offered by multiple assemblies. (In particular, java.lang.Object and java.lang.Class will be offered by each generated proxy assembly.) If more than one proxy assembly offers a proxy class, the particular class must be indicated using a fully-qualified name *ClassName*,*AssemblyName*.

Discovering information about Java sides from the .NET side

A .NET-side client program can obtain information about the Java sides with which it is communicating.

The method

```
static bool com.jnbridge.jnbproxy.JavaSides.isAlive(string name);
```

will return true if the Java side with the supplied name is currently running, and false otherwise (for example, if it failed, or has been halted). If an invalid Java side name has been supplied, a RemotingException will be thrown.

For convenience, the method

```
static bool com.jnbridge.jnbproxy.JavaSides.isAlive();
```

will return the status of the default Java side (which can also be queried with the first version of isAlive(), using the name "default".

The property

```
static string com.jnbridge.jnbproxy.JavaSides.CurrentJavaSideName { get; }
```

returns the name of the currently active Java side with which the .NET-side client is communicating.

The method

```
static com.jnbridge.jnbcode.JavaSideInfo
        com.jnbridge.jnbproxy.JavaSides.getJavaSideInfo(string name)
```

returns a JavaSideInfo object containing information on the Java side with the supplied *name*. The JavaSideInfo object contains four readable properties:

- **string JavaSideInfo.Name** returns the name of the Java side.
- **string JavaSideInfo.Scheme** returns the protocol used to communicate with the Java side. It must be *jtcp*.
- **string JavaSideInfo.RemoteHost** returns the host name or IP address of the machine on which the Java side resides
- **int JavaSideInfo.RemotePort** returns the number of the port on which the Java side is listening for requests

Discovering information about licensing

In many situations, it would be useful for an application to query the status of its JNBridgePro license. For example, an application could detect whether its JNBridgePro license is an evaluation license, and, if there are only a few days left, remind the user to purchase a valid license. To support this capability, JNBridgePro provides an API in the abstract class com.jnbridge.jnbcore.LicenseQuery, which contains the following methods:

- **static bool isValidLicense()** returns true if there is a valid license, false otherwise.
- **static bool isTimeLimitedLicense()** if the license is valid, returns true if the license is time-limited (for example, an evaluation license), false otherwise. If the license is not valid, the result is undefined.
- **static int timeLeft()** if the license is valid and time-limited, returns the number of days left. If the license is invalid, or valid but not time-limited, the result is undefined.
- **static string getInfo()** returns an descriptive string providing details about the license, useful for debugging.

Failover to a backup server (Server license only)

In many enterprise installations, a backup server or servers may exist. If the primary servers go down, the backup servers should seamlessly pick up the load. Server-licensed copies of JNBridgePro have such a failover capability. They can be configured so that if a primary server goes down, they will automatically direct all subsequent requests to a specified backup server.

Failover servers are specified in the application configuration file (*app*.exe.config for an application *app*.exe, or web.config for an ASP.NET Web application). Each Java-side, including the default, may be assigned its own failover server. Also, the failover server may use a different protocol from the primary server. Specify the failover servers in the configuration file as follows:

Inside the <configuration> section of the application configuration file, add the following section:

```
<configSections>
   <sectionGroup name="jnbridge">
   <section name="dotNetToJavaConfig"
             type="System.Configuration.SingleTagSectionHandler" />
   <section name="javaToDotNetConfig"
             type="System.Configuration.SingleTagSectionHandler" />
   <section name="tcpNoDelay"
             type="System.Configuration.SingleTagSectionHandler" />
   <section name="javaSideDeclarations"
             type="System.Configuration.NameValueSectionHandler" />
   <section name="assemblyList"
             type="com.jnbridge.jnbcore.AssemblyListHandler, JNBShare,
Version=12.10.0.0, Culture=neutral, PublicKeyToken=b18a44fb28aea122" />
   </sectionGroup>
</configSections>
```

Note that for historical reasons, for the 4.8-targeted version of JNBridgePro being used, the "assemblyList" definition above should refer to "JNBShare, Version=12.10.0.0," not "12.0.0.0".

If there is already a <configSections> section, simply add the <sectionGroup> and <section> tags above to the <configSections> section.

Inside the <configuration> section of the application configuration file, add the following section:

```
<jnbridge>
  <javaSideDeclarations>
  … new Java-side declarations go here …
  </javaSideDeclarations>
</jnbridge>
```

If there is already a <javaSideDeclarations> section, simply add the failover specifications to it.

A failover server for a Java-side named *X* on host *Host* and port *Port*, using protocol *Protocol* (jtcp, if binary/tcp) is as follows:

```
<add key="X_FailoverURL" value="Protocol://Host:Port/JNBDispatcher" />
```

For example, a failover server for the default Java-side, where the failover server is on host f, using port 8091 and binary communication would look as follows:

```
<add key="default_FailoverURL" value="jtcp://f:8091/JNBDispatcher" />
```

If there is no *X*_FailoverURL specification for a Java-side *X*, the Java-side is not configured for failover, and if *X*'s primary server fails, an exception will be thrown.

In addition to the failover server, it is possible to specify the number of times the .NET-side must try to contact the Java-side before failover occurs. For example, if the .NET-side must attempt to contact the Java-side *X*'s primary server three times before failing over, the following specification would be added to the application configuration file:

```
<add key="X_FailoverTries" value="3" />
```

If a *X*_FailoverTries specification is omitted, the .NET-side will attempt to contact the primary server for the Java-side *X* only once before failing over.

When the .NET-side attempts to contact a primary server, fails, and fails over to the failover server, it first throws a com.jnbridge.jnbcore.FailoverException. This allows the client to perform whatever actions are necessary to accommodate the failover. For example, .NET proxy objects that reference Java objects are no longer valid, since the references point to objects on the failed primary server. The appropriate action may be to create new proxy objects, which will automatically and transparently now reference Java objects on the failover server. For example, if a .NET desktop application references objects on a Java-side, it might be appropriate, on receipt of a FailoverException to close the current user session and open a new one, thereby creating new objects. If callbacks are being used, it will also be necessary to create and register new callback objects.

In addition to members inherited from System.Exception, FailoverException contains one additional public property, JavaSideName, that gets the name of the Java-side that failed over.

If the primary server is brought back up, the failover mechanism can be programmatically reset through a call to com.jnbridge.jnbcore.FailoverController.reset(string *name*), where *name* is the name of the Java-side ("default" for the default Java side). As with the original failover, new proxies must be instantiated, since the old proxies reference objects on the failover server.

Important note: Failovers can only be discovered, and FailoverExceptions thrown, by explicit calls from .NET to Java. In applications where the main contact between the .NET side and the Java side is through callback objects, FailoverExceptions are not thrown by the callback objects and it is necessary to determine whether failover has occurred through explicitly polling the Java side from the .NET side. Polling may be accomplished through either of two methods provided by com.jnbridge.jnbcore.FailoverController:

- void checkFailover(string *name*). This method checks to see whether failover has occurred for the named Java-side. (The default Java-side is named "default".) If failover has not occurred, the method will simply return. If failover has occurred, a FailoverException will be thrown. A FailoverException will only be thrown once for any given Java-side.
- Void checkFailover(). This method checks to see whether failover has occurred for any Java-side. If no failover has occurred, the method will simply return. If failover has occurred, a FailoverException will be thrown. You can see which Java-side has failed over by inspecting the JavaSideName property in the FailoverException object. A FailoverException will only be thrown once for any given Java-side.

Necessary permissions when running a .NET side

Any user accessing proxies generated by JNBridgePro must "Read" and "Write" access to the folder `<System-drive>:\Documents and Settings\All Users\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 ASP.NET is running on a machine running the NTFS file system, you must make sure that the user account running the ASP.NET program (typically ASPNET, but you may have changed it) "Read" and "Write" access to the folder
`<System-drive>:\Documents and Settings\All Users\Application Data\Microsoft\Crypto\RSA\MachineKeys`.

If the required access permission has not been assigned, the ASP.NET program will not be able to use the proxies. In addition, the account running ASP.NET must be a member of the "Power Users" group.

When using shared-memory communications with ASP.NET, you must make sure that the account running the program (typically ASPNET) has "Read & Execute," "List Folder Contents," and "Read" access permissions to the folders containing the Java class and jar files used by the programs (including jnbcore.jar and bcel-*6.n.m*.jar). If these permissions are not present, a ClassNotFoundException or a NoClassDefFoundError may be thrown when a class or jar file in the classpath cannot be read. If jnbcore.jar or bcel-*6.n.m*.jar cannot be read, a SharedMemoryException will be thrown. In addition, if the file jvm.dll specified in the "jvm" item cannot be read, an SharedMemoryException will be thrown.

Using proxies from earlier JNBridgePro versions with JNBridgePro v12.0

If your application uses JNBridgePro 12.0, it must use proxies generated by the JNBridgePro 12.0, 11.0, 10.1, 10.0 or 9.0 proxy generation tools. If you are using JNBridgePro 12.0, you cannot use proxies generated by JNBridgePro version 8.2 or earlier.

Tuning 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 the typical JNBridgePro-generated network packet is small, and in some cases this may lead to network congestion and degraded performance. If you encounter network performance degradation, you can turn off this NoDelay option so that packets are aggregated before they go out. In some cases this may improve network performance. Typically, if calls or returns contain a large amount of data (for example, if you are passing large arrays, or large by-value objects), 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 your .NET application's configuration file. That is, if your 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>
   <sectionGroup name="jnbridge">
   <section name="dotNetToJavaConfig"
              type="System.Configuration.SingleTagSectionHandler" />
   <section name="javaToDotNetConfig"
              type="System.Configuration.SingleTagSectionHandler" />
   <section name="tcpNoDelay"
              type="System.Configuration.SingleTagSectionHandler" />
   <section name="javaSideDeclarations"
              type="System.Configuration.NameValueSectionHandler" />
   <section name="assemblyList"
              type="com.jnbridge.jnbcore.AssemblyListHandler, JNBShare,
Version=12.10.0.0, Culture=neutral, PublicKeyToken=b18a44fb28aea122" />
   </sectionGroup>
   <jnbridge>
      <tcpNoDelay noDelay="false" />
 </jnbridge>
</configuration>
```

Note that for historical reasons, for the 4.8-targeted version of JNBridgePro being used, the "assemblyList" definition above should refer to "JNBShare, Version=12.10.0.0," not "12.0.0.0".

To turn the NoDelay option off in the Java-to-.NET direction, add the following line to your jnbcore.properties file

```
javaSide.nodelay=false
```

Deploying JNBridgePro as part of another application

While JNBridgePro installer can be run in "deployment configuration," this is not necessary when deploying an application that uses JNBridgePro. All that is necessary is to deploy the application including the proxy dll and the various JNBridgePro components, including jnbshare.dll, jnbsharedmem.dll (actually, jnbsharedmem_x86.dll, jnbsharedmem_x64.dll, or both, depending on whether the application is x86, x64, or Any CPU, respectively), jnbcore.jar, and bcel-*6.n.m*.jar. In addition, you must deploy either jnbauth_x86.dll or jnbauth_x64.dll, depending if your application is a 32-bit or 64-bit application. If your application has been built as "Any CPU" and can run as either a 32-bit or 64-bit application, you should deploy both jnbauth_x86.dll and jnbauth_x64.dll. jnbauth_x86.dll and jnbauth_x64.dll, when deployed, must be placed in the same folder as the application's jnbshare.dll.

Note that the licensing activities discussed below are discussed in detail in the section "Licensing," above.

If license activation is required, RegistrationTool.exe must also be deployed to the folder containing jnbshare.dll. In this case, perform license activation using the registration tool. If activation is not required and a license file is available, the file should be deployed into the application executable's folder, or into the JNBridgePro installation folder if the installer was run. If you have an activation key for an evaluation license, you can use it to obtain a 30-day evaluation license, then obtain a permanent license later. You can also run the registration tool from a script, using its command-line interface. Finally, you can add information to the application configuration file indicating the location of the license file, or of a license manager, if one is being used.

Using JNBridgePro with .NET Framework 3.5 and earlier

JNBridgePro will work with .NET Frameworks 4.0, 4.5, 4.6, 4.7, and 4.8. JNBridgePro will *not* work with .NET Frameworks 3.5 and earlier.

Running the Java side under non-default security managers

The Java side will work without problem under the default security manager (that is, the one that is 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 explicitly grant permissions for your Java classes running on the Java side.

Running the Java-side inside a JEE application server (including accessing Enterprise Java Beans)

JNBridgePro can be used to access Java code running inside any JEE application server, or inside a servlet container such as Tomcat. This includes accessing Enterprise Java Beans (EJBs) and other Java EE facilities. The recommended way of implementing this scenario is to utilize any Java client stub classes provided for the application (for example, EJB client stub classes). Proxy the client stubs, then bridge to them from the .NET program on the local (.NET client) machine using the shared-memory communications mechanism. The Java client stubs will then communicate with the remote JEE application server using the native, or vendor-specific Java communications mechanism. This approach has the advantage of not having to touch the remote application server.

However, it is also possible to deploy the Java side into the application server. While this is more complicated and requires the application server to be touched, it does allow multiple clients to access a single .NET side. Using JNBridgePro in this way requires four steps:

- First, create the proxy assembly. This process is similar to creating proxies in the ordinary way, but involves a few extra considerations.
- Second, create a WAR (Web Application aRchive) file jnbcore.war.
- Third, deploy jnbcore.war in your application server or servlet container.
- Fourth, build and run your .NET application.

Examples showing how to use JNDI and access EJBs on specific application servers from JNBridgePro can be found in the Demos\J2EE-Examples folder in your JNBridgePro installation.

Creating the proxy assembly

The procedure for creating a proxy assembly to access Java code running is identical to creating proxy assemblies in the normal way except that certain Java EE-related jar files must be included in the proxy generation class path in order to get optimal results. First, if you are accessing EJBs, you must place the jar file containing the EJBs being accessed in the proxy generation classpath. Second, any Java EE API classes you access directly from .NET (such as JNDI contexts or EJB-related exceptions) must be in the classpath. Third, the classes EJBHome and EJBObject and their supporting classes must be in the classpath. The simplest way to do this is to add the jar files containing classes in question to your proxy generation classpath. Most of the Java EE-related classes reside inside a jar file with a name such as j2ee.jar. The name of the jar file may vary from one application server to another: in JBoss it is jboss-j2ee.jar; in WebLogic 6.1 it is j2ee12.jar, and in WebSphere 4.0 it is j2ee.jar. Some application servers may place relevant classes in other jar files: for example, JBoss places JNDI-related classes into jndi.jar. Other classes, such as JMS classes, may be in other jar files, depending on the application server. These jar files are typically in a directory 'lib' immediately under the application server's root directory. To explore a jar file to determine whether it contains desired classes, use the jar tool that comes standard with the Java 2 SDK (type '`jar tf j2ee.jar`', for example, to list the classes in the file j2ee.jar in the current folder), or use the WinZip compression utility. **Be careful not to alter the contents of these jar files.**

Once the classpath has been set up, generate proxies for all classes used by your .NET code. For EJBs, this will typically include the home and remote interfaces for each EJB that is accessed from

.NET. For each class for which a proxy is to be generated, it is recommended that proxies for all supporting classes also be generated, so that proxies are also generated for all exceptions that might be thrown. This is typically done by checking the "Include supporting classes" box when performing the "Add Classes from Classpath" operation in the JNBProxy tool.

Once the classpath has been set and the classes have been loaded, build a proxy assembly in the normal manner.

Creating jnbcore.war

To use the JNBridgePro's Java-side runtime component jnbcore.jar inside a Java EE application server, it must first be packaged inside a WAR (Web Application aRchive) file with associated configuration information. Future versions of JNBridgePro will include a tool to help automate the WAR file generation process, but it is a simple matter to generate these files "by hand," by performing the following steps in order:

- Create an empty folder to contain the components of the WAR. Inside this empty "root" folder, create a folder called 'WEB-INF', and inside the 'WEB-INF' folder, create another folder called 'lib'.
- In the folder WEB-INF/lib (WEB-INF\lib if running on Windows), place copies of jnbcore.jar and any other jar files containing classes that your .NET program will be accessing through JNDI. Typically, this would include the jar files containing the EJBs your program will be accessing. Note that these EJB jar files are copies of the original EJB jar files used by the application server, which must be deployed on your application server in the customary way. If you omit the copies of the EJB jar files from your jnbcore.war file, your .NET program will not be able to locate these classes through JNDI, and ClassNotFoundExceptions will be thrown.
- In the WEB-INF folder, place your web.xml configuration file. It should contain code similar to the following:

```
<?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>
    <servlet>
        <servlet-name>JNBServlet</servlet-name>
        <servlet-class>com.jnbridge.jnbcore.JNBServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

<!-- EJB References: one for each EJB accessed from the .NET code -->
    <ejb-ref>
        <ejb-ref-name>EJB name goes here</ejb-ref-name>
        <ejb-ref-type> Session or entity type goes here</ejb-ref-type>
        <home>EJB's home interface goes here</home>
        <remote>EJB's remote interface goes here</remote>
    </ejb-ref>
</web-app>
```

The web.xml file's contents should be identical to the XML above, except that the EJB references section should contain an entry for each Enterprise Java Bean that will be accessed by the .NET code. Each entry should contain the EJB's name, type (session or entity), and home and remote interfaces. See the Demos\J2EE-Examples folder for examples of how web.xml should be constructed.

- In the WEB-INF folder, place a copy of the jnbcore.properties file specifying the protocol that the Java-side will be using, and the port on which it will be listening. The jnbcore.properties file is identical to that used when jnbcore.jar is used in standalone fashion.

  Alternatively, you can omit jnbcore.properties from the WAR file and direct jnbcore.jar to look for jnbcore.properties at a specific place in the file system by modifying the web.xml file as follows:

```
<servlet>
  <servlet-name>JNBServlet</servlet-name>
  <servlet-class>com.jnbridge.jnbcore.JNBServlet</servlet-class>
  <init-param>
    <param-name>props</param-name>
    <param-value>FULL PATH OF JNBCORE.PROPERTIES FILE GOES HERE</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

  In this configuration, the servlet tag in the web.xml file contains an init-param subtag containing the full path of the jnbcore.properties file to be used. This configuration is most useful when debugging.

- In the WEB-INF folder, add any necessary additional server-specific configuration files specifying JNDI name mappings. In JBoss, this file is called jboss-web.xml. In WebLogic, it is called weblogic.xml. See Demos\J2EE-Examples for examples of how these files are constructed.
- Construct the WAR file by creating a command-line window, navigating to the root folder containing the WAR components and subfolders, and typing the command 'jar cf jnbcore.war WEB-INF'. The file jnbcore.war will be created in this folder.

Once jnbcore.war has been created, it can be deployed to the application server or servlet container. Consult the instructions for your application server for specific information on how WAR files are deployed. When jnbcore.war is deployed, there should be an indication in the standard output window of your application server that jnbcore has started, and a binary server has been created.

Note that it is possible to have more than one instance of jnbcore.war installed; for example, if you have multiple .NET applications, each communicating with a different set of Enterprise Java Beans or a different Enterprise Application. If you do have more than one instance of jnbcore.war, each instance must have a different name (e.g., jnbcore1.war, jnbcore2.war, etc.), and each instance must be configured (through its own copy of jnbcore.properties) to be listening on a different port.

Assuming that the application server has been started and jnbcore.war has been deployed, simply build and run your .NET program in the usual way.

Note that some application servers (WebLogic, for example) support 'exploded' web applications in addition to WAR files. In such cases, the folder hierarchy containing the Web application components (before being converted to a WAR file) may be deployed directly. See the documentation of your application server for more information.


JNBPRO

Using generated proxies

Development

Once proxies have been generated, you can write .NET code that uses the proxies to call the corresponding Java classes. In order to access the Java classes during development, you must generate an assembly containing proxy classes as described above. In addition, the DLL file `jnbshare.dll` must be installed on the development machine. In the development environment, add references to the proxy assembly (in the example above, `jnbproxies.dll`) and `jnbshare.dll` to the current project. You can then call the Java classes by writing calls to the proxies. The proxies have the same name as their corresponding Java class, and reside in a namespace identical to the package in which the corresponding Java class resides. In the example above, you can make calls to the Java class Customer by making calls to the .NET proxy class Customer in the namespace com.jnbridge.samples.

For example, to create an instance of the Java class Customer, you can write the following code, using C#:

```
using com.jnbridge.samples;
…
Customer c = new Customer();
```

you can then call the proxy object as if you were directly calling the Java object:

```
int i = c.getLastOrder();
```

Static calls are also possible:

```
int j = Customer.getLastCustomerID();
```

Fields may be accessed in the normal way:

```
string s = c.name;
c.name = "new customer name";
```

Constructor calls, parameter passing, and return values all work as though the Java classes were being called directly:

```
Person p = c.getContact();
c.setContact(new Person("John Q. Public"));
```

Thrown exceptions are also caught as if you were directly catching the Java exceptions:

```
try
{
        c.addOrder("an invalid order");
}
catch (InvalidOrderException)
{
        // handle the exception here
}
```

(For a full discussion of the way JNBridgePro maps Java classes to .NET proxies, including handling of interfaces and abstract classes, see *Mapping Java entities to .NET*, below.)

You not only have the ability to use JNBridgePro to call Java classes and objects, you can also inherit from Java classes and objects. For example, assuming the example given above, you can create a .NET class InactiveCustomer in C# by inheriting from Customer as follows:

```
using com.jnbridge.samples;
…
public class InactiveCustomer : Customer
{
        dateTime dateInactivated;
        public override bool customerStatus();
}
```

In addition to the new field dateInactivated and the overriden method customerStatus(), all the functions and fields available in Customer are also available in InactiveCustomer, and constructing a new InactiveCustomer will cause invocation of Customer's constructor, as expected.

You can also pass an instance of InactiveCustomer wherever an instance of Customer is expected:

```
InvalidCustomer ic = new InvalidCustomer();
…
Customer.registerCustomer(ic);
```

There is one restriction to the use of inheritance. The .NET subclasses to the Java classes are not recognized by the Java-side, so that in the example above, if the Java static method Customer.registerCustomer() looks like the following:

```
static public void registerCustomer(Customer c)
{
        boolean b = c.customerStatus();
}
```

the customerStatus() method defined in class Customer will be called, not the overriding method in class InvalidCustomer. This restriction will be removed in a future version.

Note: When extending a proxy class, note that if a Java class does not have a default constructor, its .NET-side proxy will have a protected default constructor. Do not use this constructor: it is there for internal use only. If you use this constructor, the underlying Java class will not be instantiated, and subsequent calls to instance methods and fields of the proxy will cause a Java-side NullPointerException to be thrown.

Garbage collection and object lifecycle management

In JNBridgePro, when proxy objects are no longer reachable and are garbage collected, the finalization routine will notifying the corresponding Java side that the object no longer has an external reference and may itself be ready for garbage collection. Generally, nothing more needs to be done, and object lifecycle management is completely transparent.

However, there are some situations where it is desirable to have finer control over object lifecycle management. For example, .NET garbage collection is typically triggered by memory usage exceeding certain threshold levels. Consider a situation where a .NET proxy object represents a very large Java object, or is at the root of a very large set of Java objects. Even if the proxy object is no longer reachable, .NET garbage collection may not be triggered because there is little memory pressure on the .NET side, the Java-side objects are not released, and the Java side may run out of memory even though the Java-side objects are no longer being used.

In order to address these situations, JNBridgePro offers a mechanism for releasing Java objects before their .NET proxies are garbage collected. All .NET proxies implement the System.IDisposible interface, which means that each proxy implements the Dispose() method. If Dispose() is called on a .NET proxy, the corresponding Java object is released and may become eligible for garbage collection. If a proxy is accessed in any way after its Dispose() method is called, a System.ObjectDisposedException will be thrown. If the Dispose() method is not called on a proxy object, the underlying Java object will be released upon finalization, as before.

As a convenience, .NET-side client programs can call the method

```
static void com.jnbridge.jnbproxy.JavaSide.disposeAll()
```

to release the Java objects corresponding to all currently active .NET proxies. It is the equivalent of calling Dispose() on all currently active .NET proxies. JavaSide.disposeAll() should be used with care; typically it is useful as one of the last statements before the .NET client program terminates.

Running the application

To run the application once it has been written and built, first configure the .NET- and Java-sides as described in the section *System configuration for proxy use*, above. Then, start the Java-side as described in the section *Configuring and starting a standalone JVM for proxy use*, above. Finally, start the .NET-side application.

Data compression

The tcp/binary communications mechanism in JNBridgePro now compresses binary messages over a certain size before exchanging them between the .NET and Java sides. This process is transparent to the user, and ordinarily nothing needs to be done about it. Please note that to prevent the Java side from returning compressed messages, the javaSide.noCompress property can be set to "true". If it is missing, or set to anything else, the Java side will return compressed messages.

Multiple proxy DLLs

Prior to JNBridgePro 9.0, only one proxy DLL was allowed in .NET-to-Java projects. Starting with JNBridgePro 9.0, multiple proxy DLLs may be used in a project.

When using multiple proxy DLLs, a proxy defined in one DLL may not be passed to a proxy defined in a different DLL. That is, a proxy defined in one DLL may not be passed as an argument to a method or assigned to a field defined in a proxy from a different DLL. This should be caught at compilation time, since proxies from different proxy dlls are of different types, even though they have the same name, and proxies from different proxy dlls are in separate inheritance hierarchies.

In order to disambiguate situations where proxies of the same name are defined, Visual Studio provides a mechanism for distinguishing between references to the two proxy types. Once the proxy dlls are referenced in your Visual Studio project, access the properties of each proxy dll (that is, select the proxy dll under references, right-click on the referenced proxy dll, and select **Properties**). In the Properties pane, locate the **Aliases** property, and enter a unique name as the proxy dll's alias. Now, you can use the alias as part of the proxied class's name.

For example, assume there are two proxy dlls in the project: proxy1.dll and proxy2.dll. In Visual Studio, using the instructions above, assign proxy1.dll the alias *proxy1*, and assign proxy2.dll the alias *proxy2*. Both proxy dlls will have a proxy for java.lang.Object, and if you simply refer to java.lang.Object, Visual Studio will report an ambiguous reference since the class appears in both

dlls. You can disambiguate the reference by adding the following lines to the beginning of your source file:

```
extern alias proxy1;
extern alias proxy2;
```

You can then refer to the java.lang.Object in proxy1.dll by referring to proxy1::java.lang.Object or proxy1.java.lang.Object. Similarly, you can refer to the java.lang.Object in proxy2.dl by referring to proxy2::java.lang.Object or proxy2.java.lang.Object.

Note that the above mechanism is only supported by C#. Is is not available in VB.NET.

Also note that, if a proxy of the same name appears in two or more proxy DLLs, and there is only one Java side, then the multiple identically-named proxies will each reference the same underlying Java class, which will be the first class of that name encountered in the classpath. If your multiple identically-named proxies must reference different underlying Java classes (for example, if they are in different jar files), then you should place the libraries with the identically-named classes in different Java sides. See the section "Interacting with multiple Java sides" for more information.

Embedding Java GUI elements inside .NET GUI applications

It is possible to embed Java GUI elements, including AWT and Swing elements, inside Windows Forms applications and Windows Presentation Foundation (WPF) applications. To do so, the Java GUI element (which must be derived from java.awt.Component), must be wrapped inside a com.jnbridge.embedding.JavaControl (when embedding inside Windows Forms) or com.jnbridge.embedding.JavaWPFControl (when embedding inside WPF) object as follows:

```
MyJavaComponent mjcomp = new MyJavaComponent();      // MyJavaComponent is a proxy of a
                                                     // Java GUI component
JavaControl jc = new JavaControl(mjcomp, component_width, component_height);
or
JavaWPFControl jc = new JavaWPFControl(mjcomp, component_width, component_height);
```

The JavaControl object is derived from System.Windows.Forms.Control, and can be used wherever any other WinForms control can be used. Similarly, the JavaWPFControl is derived from System.Windows.UIElement and can be used wherever any other WPF UIElement can be used.

Note that the JavaControl and JavaWPFControl constructors require that the embedded component's width and height, in pixels, be specified.

Note: Use of JavaWPFControl requires .NET Framework version 3.0 or later. If .NET Framework 2.0 (or earlier) is used with JavaWPFControl, an exception will be thrown.

To use JavaWPFControl, your application must reference jnbwpfembedding.dll, and the file must be deployed into the Global Assembly Cache (GAC), or into the folder in which the application will run.

Any listeners on events in the Java component can be registered directly with the Java component proxy object using the usual callback mechanism.

Embedding Java GUI elements inside .NET GUIs can only be done when using shared memory communications.

jawt.dll and JNBJayaEntry.dll must be placed in the folder in which the application will run. If they are not there, an exception will be thrown.

Currently, embedding SWT components inside .NET GUIs is not supported.

Note: In addition to the standard Java-in-.NET embedding feature, we have provided a new "experimental" reimplementation to work with Java 7 and later. Changes to the native focus implementation in Java 7 broke the handling of focus-related events, particularly keyboard events, in the embedded Java controls. This new implementation, called com.jnbridge.embedding.NewJavaControl, restores these focus-related capabilities. There are a few caveats:

- ***Only embedding of Java controls inside Windows Forms is currently supported. Embedding Java inside Windows Presentation Foundation (WPF) will be supported in a future release.***
- ***There may be some unresolved issues related to z-order (that is, in certain situations, the Java control might appear in front of an element that it should not appear in front of, or behind an element that it should not be behind). Please report such issues if you encounter them.***

The original implementation properly handles focus-related events in Java 5 and 6, and properly handles embedding (except for focus-related events) in Java 7 and later.

For more information, please see the JNBridgePro knowledge base at https://jnbridge.com/support/knowledge-base.

For an example of embedding, please see the "Embedding Java in .NET GUIs" demo that comes with the JNBridgePro distribution.

Mapping Java entities to .NET

JNBridgePro applies a set of mapping rules in order to map Java class APIs to their .NET equivalents. Although these rules are generally straightforward, they are not strictly one-to-one, due to differences between the Java and .NET platforms.

Directly mapped classes

Java primitive classes are directly mapped to .NET classes as follows:

Java	.NET	Comments
int	int (System.Int32)	
short	short (System.Int16)	
long	long (System.Int64)	
float	float (System.Single)	
double	double (System.Double)	
char	char (System.Char)	
byte	sbyte (System.Sbyte)	The rationale for this mapping is that the Java byte type is signed, while the .NET byte type is unsigned.
boolean	bool (System.Boolean)	

In addition, the class java.lang.String is directly mapped, for efficiency reasons, to the .NET class string (System.String).

Arrays of any rank or dimension, whose base elements are primitives or strings are also directly mapped to .NET arrays, for efficiency reasons. Thus, a Java method that returns a value of, say, String[][] will be represented in the .NET proxy by a method that returns a .NET array of type string[][], where string is the .NET class string.

To pass or return a string, enum, or array where a proxy method or field expects a java.lang.Object, you must wrap it in a JavaString, JavaEnum, or JavaArray class. See *Passing strings, enums, or arrays in place of objects*, below.

A number of Java and .NET non-primitive are directly mapped to each other in order to improve access times. See *Enums*, *Directly mapped collections* and *Other directly mapped value objects*, below.

Enums

JNBridgePro offers the choice of mapping Java enum classes directly to .NET enum classes, or to simply proxy them and treat them as ordinary Java classes. The JNBridgePro proxy generation tool provides an option (a check box in the Java Options dialog box of the GUI-based standalone tool or the Visual Studio plug-in, or the /dme option in command-line tool) to determine which mapping behavior to use. See the section "Proxying Java enums" for more details.

JNBridgePro's default behavior is that Java enum classes supported by Java 5.0 are mapped directly to .NET enum classes and values. When a Java enum class is proxied, the proxy assembly will contain a .NET enum of the same name and containing the same set of enum values. If a Java method returns an enum, the equivalent proxy method will return the corresponding .NET enum, and if the method expects an enum as a parameter, the proxy method will expect the corresponding .NET enum as a parameter. Similarly for setting and getting fields: if a field's type is a Java enum, the proxy's corresponding field will expect the corresponding .NET enum.

To pass or return a mapped enum where a proxy method or field expects a java.lang.Object, you must wrap it in a JavaEnum class. See *Passing strings, enums, or arrays in place of objects*, below.

When the option to map Java enums to .NET enums is turned off, Java enums are proxied and handled in the same manner as any other proxied Java class. In this case, enum proxies do inherit from java.lang.Object, and they do not need to be wrapped in a JavaEnum class.

Note: Java enum classes can have associated behaviors as methods. Since .NET enums cannot have corresponding associated behaviors, the Java enums' methods are not proxied when Java enums are mapped to .NET enums. These methods are proxied when mapping is turned off.

Note: Java allows constant-specific class bodies to be attached to the enum constants to specify behaviors. For example, we can define the following Java enum:

```
public enum Operation
{
        PLUS { double eval(double x, double y) { return x + y; } },
        MINUS { double eval(double x, double y) { return x - y; } };
}
```

This class associates different behaviors with the enum constants PLUS and MINUS. The Java compiler typically generates anonymous subclasses of Operation for each class body (for example, Operation$1 and Operation$2). Since .NET has no equivalent construct, we do not proxy the anonymous subclasses, and the constant-specific class bodies are not available from .NET when Java enums are mapped to .NET enums. The constant-specific class bodies are proxied when mapping is turned off.

Arrays

Arrays passed from the Java to the .NET side are automatically converted from Java arrays to .NET arrays. Similarly, .NET arrays passed to Java are converted from .NET arrays to Java arrays. This is done for reasons of efficiency: if a Java method only returned an array reference to the .NET side, rather than the entire array, then every access of an individual array element through a subscripting operation would involve a JNBridgePro round trip from the .NET side to the Java side and back. This would involve unacceptable overhead. By converting the entire array from Java to .NET, all element accesses are local, and much more efficient.

Generally, this mechanism manipulates arrays as one would expect. However. copying arrays between Java and .NET (that is, passing them *by value*), can lead to two cases of unexpected behavior of which the user should be aware. The first has to do with changes to arrays that have been passed as parameters. Given a Java method

```
void f(int[] a)
{
  for (int i = 0; i < a.length; i++) a[I] = 0;
}
```

a call to the Java method as follows

```
int[] intArray = new int[]{1,2,3};
f(intArray);
```

will result in all the elements of array being zeroed out after the return from the call to f(). The behavior in the equivalent C# program is identical. The reason for this is that a *reference* to the array, not a copy of the array, is passed to f(), so that the parameter a in f() is the same array as intArray, and all changes to a are reflected in intArray. Since JNBridgePro passes the arrays by value, any changes to a in the method f() are *not* reflected in intArray after f() returns. The user should be aware of this behavior.

The second aspect of unexpected behavior concerning arrays occurs in assigning elements to arrays that are fields of proxy objects. Consider, for example, the following Java class:

```
public class C
{
  int[] intArray = int[3];
}
```

If a .NET proxy for C is created, the following assignment statement will *not* cause a change to the corresponding Java object's intArray:

```
C c = new C();
for (int i = 0; i < c.intArray.Length; i++)
{
  c.intArray[i] = i;
}
```

The reason for this is that "fields" in proxy classes are really properties that manage the mechanism that accesses the fields in the underlying Java classes. These properties retrieve the field values through get methods, and store new field values through set methods. In the case of

```
c.intArray[i] = i;
```

c.intArray's get method executes to return a *copy* of intArray, and the assignment object changes element i of that copy to i. The original Java-side intArray is *not* changed.

To guarantee that the Java-side intArray is changed, we need to rework the .NET code so that c.intArray's set method is guaranteed to execute:

```
C c = new C();
int[] intArray = c.intArray;          // copy the array to .NET
for (int i = 0; i < intArray.Length; i++)
{
  intArray[i] = i;            // change the element of the local copy
}
c.intArray = intArray;                // copy the array from .NET back to Java
```

In the last statement, c.intArray = … forces c.intArray's set method to execute, and the underlying intArray field in the Java object will be changed. In addition to providing the expected behavior, this code is much more efficient, since it involves fewer round trips and causes the array to be copied between the .NET and Java sides fewer times.

Note that these two relatively uncommon cases (assignment to arrays passed as parameters, and assignment to elements of arrays that are fields of proxy classes) are the only situations where unexpected behavior occur because arrays are passed by value. In all other cases, assignment to arrays behaves as expected.

Class-name mapping algorithm

A proxy class has the same name as the corresponding Java classes, and each proxy resides in a namespace whose name is identical to the package the underlying class resides in. For example, the Java class java.util.Vector (class name Vector, package name java.util) is mapped to the .NET proxy java.util.Vector (class name Vector, namespace name java.util).

If the .NET proxy corresponding to a Java class has not been generated (for example, if the /ns option has been chosen in jnbproxy, or if a Java call returns an object of an anonymous or dynamically generated class for which a proxy could not be generated in advance), a proxy for the object is generated dynamically and on the fly. This new proxy can be cast to any superclass it extends, or to any interface that it implements, and any method supported by the superclass or interface to which it has been cast can be called. This provides the complete class cast capability offered by both .NET and Java; with the previous proxy substitution scheme, certain class casts were not possible.

Proxy-mapped classes

All Java classes, other than the directly mapped classes listed above, are mapped to generated proxy classes using the class-name mapping algorithm.

The Java value null is directly mapped to the .NET value null.

Access attributes of Java classes are mapped directly to .NET access attributes, although the mapped attributes may not have identical meaning. Public Java classes are mapped to public.NET proxy classes. Non-public classes are not mapped to .NET proxies.

The superclass of a Java class is mapped to the superclass's .NET proxy class using the class-name mapping algorithm, described above.

Similarly, all of a class's interfaces are mapped to proxy interfaces of the same name. If a proxy interface has not been generated, the interface is not mapped.

For each field in a Java class, a field (actually, a property, but use of a property looks exactly like use of a field) of the same name is placed in the generated proxy. Public and protected access attributes of the field are mapped to public and protected access attributes of the corresponding property in the proxy. Private and default access fields are not mapped to the proxy. Java fields that are static are made static in the .NET proxy. Volatile, transient, and strictfp attributes are not mapped. The class of the field is mapped using the class name mapping algorithm.

For each method and constructor in a Java class, a method or constructor of the same name is placed in the generated proxy. Public, protected, and static access attributes of the method or constructor are mapped to public, protected, and static attributes of the corresponding method or constructor of the proxy. Private and default access methods and constructors are not mapped. Methods that are static and/or abstract in the Java class are made static in the .NET proxy. Classes of method return values are mapped according to the class name mapping algorithm. *If a method or constructor has a parameter or a class for which a proxy has not been generated, that method or constructor will not be included in the generated proxy.*

Java SE 5.0 and later only: If a method or constructor in the underlying Java class has a variable number of arguments(that is, if it is of the form f(int… x)), then the corresponding proxy method will also have a variable number of arguments (that is, will be of the form f(params int[] x)).

Java 8 and later only: If the underlying Java class has been compiled using the -parameters option (available in Java 8 and later), and the proxy generation tool's Java side is Java 8 or later, then the proxy's methods and constructors will contain metadata assigning their parameters the same names as the parameters of the underlying methods and constructors. These parameter names will then appear in Visual Studio's IntelliSense when creating applications using the methods and constructors, and the parameter names will also be accessible through reflection. If earlier versions of Java are used, or the -parameters option is not used, then default parameter names will be used instead.

As a convenience, calls to the method ToString() in the generated .NET proxies (a method supported by all .NET objects including the proxies) are automatically passed through to the Java classes' toString() method, thereby returning the underlying Java object's string representation, rather than the string representation of the proxy object. As before, toString() can still be called directly through the proxies.

Interfaces

Java interfaces are mapped into .NET interfaces. Interfaces are mapped in the same way as classes, with two exceptions. First, the class name mapping algorithm is modified so that if no resulting interface is found as a result of following the superinterface chain, the interface is not mapped. This is done because interfaces are not rooted in `java.lang.Object` and so a result is not guaranteed. If such a result occurs, it is recommended that you regenerate the proxies to include all required interfaces.

Second, if a Java interface *I* contains a static final field (a constant), the field is not mapped to the proxy interface, but is moved to a specially generated .NET proxy class *IConstants*, which is an abstract class with the same access attributes as the interface. The fields have the same access status as the original interface fields, and are constants. This is done because C# does not allow fields in interfaces, while Java does.

Starting with Java 8, Java permits *static methods* in interfaces. Since C# does not permit static methods in interfaces, when a Java interface *I* with such a method is proxied, the method is moved to the special *IConstants* proxy class that is always generated to contain static fields and methods from the original Java interface.

In addition, starting with Java 8, Java permits *default methods* in interfaces. If an interface has a default method, any implementing class does not need to implement the method, in which case the default method is used. Since default methods do not exist in C#, and any class implementing an interface has to account for all the methods in the interface, the proxy of a Java class implementing an interface with a default method will have a proxied implementation of the method, whether or not the underlying Java class actually implements the method.

Generics

When generating a .NET-side proxy from a Java generic class, the proxy is actually generated from the generic's *raw class*. The raw class is obtained by removing the generic's type parameters, and by replacing all instances of the type variables with the variables' upper bounds. If any members of the class have types or parameters which are themselves generic types, they are also replaced with raw types. For example, if a proxy is generated for the following Java generic class:

```
public class GenericClass<T extends Number, U>
{
        U aField;
        ArrayList<Integer> anotherField;

        T aMethod(Collection<Float> c) { … };
}
```

the raw class will be

```
public class GenericClass
{
        Object aField;     // U's upper bound is Object
        ArrayList anotherField;

        Number aMethod(Collection c) { … };
}
```

and the proxy will be generated from the raw version of GenericClass. In other words, the .NET-side proxy of a Java generic class will not itself be generic.

Similarly, if a proxy is generated for an otherwise non-generic class with a generic method, the proxy is generated using the method in its raw form. For example, for a generic method

```
public <T extends Number> T aMethod(T t) { … }
```

the proxy is generated for

```
public Number aMethod(Number t) { … }
```

Note: The reason that the raw, non-generic versions of generic classes and methods are used for proxies and interop is that Java (Java SE 5.0 and later) uses *erasure* to implement and define the semantics of generics. Erasure is a compile-time technique in which the generic type is compiled in its raw form, static type checking is used to make sure that every use of the generic type is type-safe, and run-time type checks and casts are automatically inserted in the compiled code where necessary. The consequences for .NET/Java interoperation are that the information about specific invocations of the generic types has already been removed from the compiled code by the time the proxies are generated and used, and the actual values of the type variables for a given invocation of the generic type are no longer available (for example, we don't know if a method on the Java side has returned an instance of GenericClass<Integer, String> or GenericClass<Float, int[]>. Fortunately, the raw class is general enough to handle any potential invocation of the generic class.

It is possible to use a proxy of a generic class in such a way as to cause a ClassCastException. Consider, for example, the following Java class:

```
public class C
{
        public static ArrayList<Integer> getArrayList() { return new ArrayList<Integer>(); }

        public static void processArrayList(ArrayList<Integer> al)
        {
            Integer i = al.get(0);
        }
}
```

The .NET-side proxy of C will use the raw java.util.ArrayList class, and the following C# code is legal but will cause an exception to be thrown:

```
// assume proxies for java.util.ArrayList, java.lang.Float, and C
ArrayList al = C.getArrayList();
al.add(new java.lang.Float(3.0F));        // legal, even though getArrayList()
                                          // returned ArrayList<Integer>
C.processArrayList(al);                   // will throw ClassCastException
                                          // since Integer was expected
```

If proper care is taken, these exceptions can be avoided.

Abstract classes

Java abstract classes that implement interfaces do not have to account for all the methods in the interfaces they implement, although concrete classes based on them do. For example, the following is correct Java:

```
public interface i
{
        public int f();
}

public abstract class c implements i
{
}
```

It is not, however, correct in the .NET framework, since c is required to have a definition (even an abstract definition) of f(). To address this issue, JNBProxy "flattens" the definition of an abstract class by making sure that it accounts for all abstract methods in superclasses and interfaces. This should not have an impact on the use of the proxies, but you may notice that some methods appear in the proxy of an abstract class that were not in the corresponding Java class.

Previous to JNBridgePro v9.0, it was not permitted to have a .NET class that inherited from a proxy of a Java abstract class: if such a .NET class was instantiated, JNBridgePro would attempt to instantiate the underlying Java abstract class, and an InstantiationException would be thrown. Beginning with JNBridgePro v9.0, this no longer occurs, and it is permitted to write a .NET class that inherits from a proxied Java abstract class.

Cross-platform overrides

When a Java class is proxied to the .NET side, one can create .NET subclasses that inherit from the proxied Java class. It is possible to pass objects of that .NET subclass back to the Java side, since JNBridgePro considers them to be of the proxied type. On the Java side, any calls to methods that were overridden in the .NET subclass will be redirected to the .NET-side code in the .NET subclass. (Note that this behavior was introduced in JNBridgePro v9.0. In previous JNBridgePro versions, calls to overridden methods were not redirected back to the .NET side.)

Only methods that are considered overridable in Java (that is, non-private instance methods that are not marked as final) can be overridden in .NET-side subclasses of proxy classes.

To make use of this override functionality, any .NET class containing methods overriding proxied methods must contain a call to com.jnbridge.jnbcore.Overrides.registerOverrides() inside its static constructor. If there is no static constructor in the .NET subclass, you should add one. Ideally, the call to Overrides.registerOverrides() should be the first call in the static constructor.

Note that, if you call Overrides.registerOverrides() in a class's static constructor, then any instance of this class will by default not be garbage collected before the end of the program. The reason for this is to avoid situations where the .NET object is garbage collected before an overriden method is called on the Java side. If you are sure that your .NET object is no longer needed, and that it is safe to garbage collect it, you can call com.jnbridge.jnbcore.Overrides.unregisterOverrideObject(object), passing as a parameter the object that can now be garbage collected. Note that it is always safe to call unregisterOverrideObject() more than once on an object, or when the object is not an override object. Calling unregisterOverrideObject() with a null argument will result in an exception.

As an example of using overrides, consider the following example:

```
// Java code
public class JavaBaseClass
{
        public void method1()
        {
                System.out.println("JavaBaseClass.method1");
        }

        public void method2()
        {
                System.out.println("JavaBaseClass.method2");
        }
}

public class JavaFrameworkCode
{
        private static JavaBaseClass myClass = null;

        public static void registerObject(JavaBaseClass jbc)
        {
                myClass = jbc;
        }

        public static void runFramework()
        {
                myClass.method1();
                myClass.method2();
        }
}

// C# code
public class DotNetSubClass : JavaBaseClass
{
        static DotNetSubClass()
        {
                com.jnbridge.jnbcore.Overrides.registerOverrides();
        }

        public override void method1()
        {
                Console.WriteLine("DotNetSubClass.method1");
        }
```

```
        public override void method2()
        {
                Console.WriteLine("DotNetSubClass.method2");
                base();  // call JavaBaseClass.method2()
        }
}
```

When we run the following code from the .NET side:

```
JavaBaseClass jbc = new DotNetSubClass();

JavaFrameworkCode.registerObject(jbc);

JavaFrameworkCode.runFramework();
```

we will see the following output:

```
DotNetSubClass.method1
DotNetSubClass.method2
JavaBaseClass.method2
```

Callbacks

It is possible to create .NET callback classes that are called from the Java side when certain events occur. Callbacks are used for event handling, and may be used to update a .NET GUI or perform other .NET processing when certain events occur. Although callbacks in JNBridgePro were designed to support Java's Event Listener model, they may be used in other ways.

Any .NET class that will be used as a callback must fulfill two requirements. It must implement at least one Java-side interface (a "listener interface") for which a .NET proxy has been generated, and it must be decorated with the [Callback()] custom attribute (whose full name is com.jnbridge.jnbcore.CallbackAttribute, but which may be referred to as simply [Callback()] if the namespace com.jnbridge.jnbcore has been imported). For example, if there is a Java interface

```
package myEventHandlers;

public interface FooEventListener
{
        void fooOccurred(int fooInfo);
}
```

for which a .NET proxy has been generated, one can create a callback FooEventHandler, which handles FooEvents, in C# as follows:

```
using com.jnbridge.jnbcore;
using myEventHandlers;

[Callback("myEventHandlers.FooEventListener")]
public class FooEventHandler : FooEventListener
{
        public void fooOccurred(int fooInfo)
        {
          // process foo event here
        }
}
```

Note above that the [Callback()] attribute contains as a parameter the name of the listener interface the callback class implements. If it implements more than one listener interface, it may have more than one [Callback()] attribute. ***The parameter must be the fully qualified name of the listener interface. If the interface is nested inside another Java class, you must use the special internal Java nesting delimiter in the name. For example, if the listener interface L is nested inside the Java class a.b.C, you must use the parameter "a.b.C$L", not "a.b.C.L". If you do not use the nesting delimiter, an exception will be thrown.***

A callback is typically registered by passing it to a Java-side class or object that contains a registration method that accepts parameters that implement the specified listener interface. For example, if there is a Java-side class FooEventGenerator for which a .NET proxy has been generated, with the static method registerListener() that takes as a parameter a callback object that implements the FooEventListener interface, one can register FooEventHandler through the following C# code:

```
FooEventGenerator.registerListener(new FooEventHandler());
```

Later, when a Foo event occurs, FooEventGenerator will call the fooOccurred() method on all registered listeners, including the .NET callback object of class FooEventHandler.

.NET callbacks have the following restrictions:

- All parameters of methods in the implemented listener interfaces must be of types recognized by the Java side. These include primitives (int, long, short, char, sbyte, bool, float, and double), strings, Java objects for which .NET proxies have been generated, and arrays of any of the above.
- Values returned by callback methods must be of types recognized by the Java side. These include primitives (int, long, short, char, sbyte, bool, float, and double), strings, Java objects for which .NET proxies have been generated, and arrays of any of the above.
- Exceptions thrown by callback methods must be of types for which a .NET proxy class has been generated. If any other exception is thrown, the result is undefined.
- Callback objects may not be returned to the .NET side as a result from a call to a Java method, or retrieved by the .NET side from a Java field. This restriction will be lifted in a future version.
- Callback objects may not be passed to the Java side where an object of type java.lang.Object or any other type of Java object is expected, since they do not inherit from java.lang.Object. They may only be passed where an object that implements the same listener interface is expected. This restriction will be lifted in a future version.
- Callback methods may not be declared as explicit interface implementations. That means that, in the above example, one cannot declare inside FooEventHandler:

  ```
  void myEventHandlers.FooEventListener.fooOccurred(int fooInfo) { }
  ```

  Rather, one can only declare:

  ```
  public void fooOccurred(int fooInfo) { }
  ```

  If a callback method is declared as an explicit interface implementation, an exception will be thrown at runtime.

Although the .NET callback facility has been designed to support the Java event listener model, they may be used in other ways. The only requirements are that the callback objects implement a Java interface for which a proxy has been generated, that they be annotated with the [Callback()] attribute,

and that instances of the callback classes be passed to the Java side through a method that expects objects that implement the Java interface implemented by the callback object and mentioned in the [Callback()] attribute. On the Java side, callback objects can be stored in a variable or collection, or invoked, in the same manner as any other object that implements the callback interface.

Important Note: Each callback runs in its own thread distinct from any other threads in the .NET-side application. This can lead to a number of potential pitfalls of which the developer should be aware:

- ***When a Java method fires a callback method, the Java method's thread suspends until the callback returns. If that Java method was directly or indirectly called by a .NET method, that .NET method's thread also suspends, since it is waiting for the Java method to return. The calling Java and .NET methods' threads will resume when the callback returns. This is done to preserve the conventional Java semantics. However, if the calling .NET thread is the main thread of a Windows Form, and the callback attempts to access controls in that Windows Form, deadlock may result, since the calling thread, which is suspended, needs to handle those changes to the form. To avoid this problem, if a call from .NET to Java can result in a callback, and the callback accesses controls in a Windows Form, the developer should make sure that that call is in a thread other than the form's main thread. If it is not, a new thread should be created to make the call. If this is not possible, asynchronous callbacks should be used (see below).***
- ***If a callback manipulates controls in a Windows Form, it should use the Control.Invoke method to perform the manipulations from the callback, since the callback is running in a different thread from the one in which the WinForm was created.***

Asynchronous callbacks

One way to avoid the possibility of deadlocks as described in the previous section is to use an *asynchronous callback*. Asynchronous callbacks are exactly like regular callbacks except that the Java thread invoking the callback does not suspend until the callback completes, and does not wait to see if a value is returned or an exception is thrown back. This means that .NET calls to Java methods leading to callbacks never suspend and deadlock is not a possibility.

Asynchronous callbacks cannot return values or throw exceptions back to the Java side. Any values returned by callback methods are ignored, and any exceptions thrown by callback methods are also ignored. Since callbacks can themselves call Java methods, this can be used to get the same effect as returning values to the Java side directly.

To designate an asynchronous callback, a class must be annotated with the [AsyncCallback()] (com.jnbridge.jnbcore.AsyncCallbackAttribute) custom attribute. This attribute is used in the same way as [Callback()]; it takes one string parameter giving the name of the listener class it implements.

A class may not have both [Callback()] and [AsyncCallback()] attributes. If a class with both types of callback attributes is passed to the Java side, a com.jnbridge.jnbcore.InvalidCallbackAttributeException will be thrown.

Managing callback resources

Each callback object that is registered with the Java side consumes resources, including a listener thread. Even when a callback object is unregistered and the object is garbage-collected, these resources still persist, unless explicit action is taken. If an application only uses a few callbacks, this should not matter, but if the application creates many callbacks, the application may run out of resources unless they are explicitly managed.

To manage callback-related resources, we have provided an API: com.jnbridge.jnbcore.Callbacks.releaseCallback(object callbackObject).

After the callback object has been unregistered with the Java side and will no longer be used as a callback, the user can call Callbacks.releaseCallback() with the callback object as its argument. This will release the resources associated with the callback, and will avoid resource-related problems when many callbacks are created.

Another convenient way to use releaseCallback() is to have the callback object implement IDisposable, then place a call to releaseCallback() in the Dispose() method.

The callback object cannot be used as a callback after releaseCallback() has been called with it as a parameter. If the object is used as a callback after that point, an exception will be thrown. It is still possible to use the object in other ways (other than as a callback) after releaseCallback() is called.

Terminating an application with callbacks

In older versions of JNBridgePro, it was necessary to explicitly terminate callback listener threads by calling Callbacks.terminateCallbacks() upon exit from an application using callbacks, or the application would hang. Callback listener threads are now automatically terminated on application exit; it is no longer necessary to call Callbacks.terminateCallbacks(), although it does no harm to do so.

Nested Classes and Interfaces

JNBridgePro supports access of Java nested classes and interfaces from .NET. If the user adds a Java class to the JNBProxy environment, and the class has nested classes or interfaces, the nested classes are also added. This process is done recursively, so if the nested classes themselves have nested classes, those further nested classes are themselves added.

It is also possible to explicitly add a nested class. For example, if the class A encloses a class B, one can explicitly add A.B by requesting that JNBProxy add the class A$B. ('$' is the internal Java delimiter between enclosing and nested class names.) If a nested class is added, its enclosing class or interface is also automatically added, as well as any classes or interfaces that nest within it. Again, the process is recursive, and proceeds until a fixpoint is reached.

.NET nested classes do not work in exactly the same way as Java nested classes, and so they do not map directly. A non-static Java nested class (also known as an "inner" class), has a special relationship to the enclosing class, which means that an inner class must always be instantiated in the context of an enclosing class. For example in the Java code

```
public class Outer
{
        public class Inner
        {
        }
}
```

instances of the outer and inner classes must be generated as follows:

```
Outer o = new Outer();
Outer.Inner i = o.new Inner();
```

JNBridgePro creates .NET proxies that mirror the nesting structure of the Outer and Inner classes, but which require the enclosing object to be explicitly given. The .NET proxies generated for the above Outer and Inner classes is equivalent to the following C# classes:

```
public class Outer
{
        public class Inner
        {
          public Inner( Outer enclosing ) {…}
        }
}
```

For each constructor in the original Java inner class, a constructor in the .NET inner class is created that has as its first argument the enclosing object, followed by the arguments in the original Java constructor. For example, a constructor in a Java inner class with a single int argument would correspond to a constructor in a .NET inner class with two arguments: the first representing the enclosing object and the second representing the int argument in the original Java constructor.

To instantiate the inner class through the .NET proxies, use code like the following C# code:

```
Outer o = new Outer();
Outer.Inner I = new Outer.Inner(o);
```

Elements in the outer class must be accessed through o; they cannot be accessed through i.

For static nested classes, classes nested inside interfaces, and interfaces nested inside classes, the mapping from Java classes to .NET proxy classes is one-to-one.

Class Literals

To easily access a class's Class object, Java provides the notion of a *class literal*. For example, the expression C.class evaluates to class C's Class object. With JNBridgePro, something similar is available on the .NET side. If a proxy (also known as C) for the Java class C has been generated, then the .NET code C.JavaClass will evaluate to the .NET proxy for the Java class C's class object.

Since JavaClass is implemented as a property, and interfaces do not permit implementations of class members such as properties, the procedure for accessing the Class object of an interface is slightly different. To get the Class object for a Java interface I, use the expression IConstants.JavaClass, where IConstants is the .NET proxy class containing any constants defined in I. IConstants.JavaClass evaluates to the .NET proxy for the Java interface I itself.

The C.JavaClass construct only works when a proxy for class C has been generated. A call to Class.forName("C") will always work for any class that is in the current class path, including classes for which proxies have not been generated.

Exceptions

Exceptions can be thrown from the Java side to the .NET side. To throw an exception across the Java-.NET boundary, make sure that proxy classes have been generated for each exception that might be thrown. Then, when a Java exception is thrown, the .NET code can catch the corresponding proxy exception object.

An exception that is declared as thrown by a method is considered to be a supporting class for the method and for the owning class, so specifying that proxies for supporting classes be generated will result in proxies for exceptions being generated. Also, when proxies for supporting classes are generated, proxies for the set of unchecked exceptions (exceptions such as ArithmeticException that do not need to be explicitly caught or thrown where they may be caught) are automatically generated. It is also possible to explicitly specify that a proxy for an exception class should be generated.

If a proxy was not previously generated for a thrown exception, the nearest superclass for which a proxy has been generated will be used, up to the proxy for java.lang.Object, which is always generated.

To access the exception message or stack trace associated with the thrown Java exception, access the string values in the properties Message and StackTrace in the .NET proxy for the thrown exception. The StackTrace property contains stack trace information as for both the .NET side and the Java side. For compatibility with .NET client code written for earlier versions of JNBridgePro, the exception message created on the Java side can be obtained from the proxy object's JavaExceptionMessage property, and the Java-side stack trace information can be obtained from the proxy object's JavaStackTrace. (The Java-side exception message can also be obtained by calling the proxy object's getMessage() method.) .NET proxies for Java exception classes are subclasses of the System.ApplicationException class.

Proxy implementation of equality and hashcode methods

JNBridgePro maps .NET-side proxies' Equals() and GetHashCode() methods to the corresponding Java objects' equals() and hashCode() methods. This allows them to behave as expected when added to .NET hashtables and other collections.

Passing strings, enums, or arrays in place of objects

For reasons of efficiency and convenience, generated JNBridgePro proxies translate returned Java String objects into native .NET strings. Similarly, returned Java array objects are translated into native .NET arrays, and returned Java enums are translated into native .NET enums. Conversely, users pass proxies native .NET strings, enums, and arrays when the underlying Java methods and fields expect Java strings and arrays.

When a method is declared to return an Object (that is, a java.lang.Object), and the method actually returns a String, it is not possible to automatically convert the Java String to a .NET string because the .NET string is not a subclass of java.lang.Object. In such cases, the method returns an object of class java.lang.JavaString, which is a subclass of java.lang.Object and which wraps the native .NET string. To obtain the native .NET string, access the JavaString property StringValue.

For example, assuming the following Java class:

```
public class c
{
   public String retString() { return "abc"; }
   public Object retObj() { return "def"; }
}
```

.NET code would access a proxy for c as follows:

```
c o = new c();
string s = c.retString();   // s contains .NET string "abc"
java.lang.JavaString js = (java.lang.JavaString) c.retObj();
string s2 = js.StringValue;  // s2 contains .NET string "def"
```

When a method declared to return a Java Object returns an array, the proxy will return an object of class java.lang.JavaArray, a subclass of java.lang.Object. The JavaArray object wraps the native .NET object, which can be retrieved by accessing the property ArrayValue. ArrayValue contains on object of .NET class System.Array, which is the base class for all .NET arrays. Therefore, assuming the following Java class:

```
public class d
{
  public int[] retArray() { return new int[3] { 1, 2, 3 }; }
  public Object retObj() { return new int[3] { 4, 5, 6 }; }
}
```

.NET code would access a proxy for d as follows:

```
d o2 = new d();
int[] a = d.retArray();   // a contains .NET array { 1, 2, 3 }
java.lang.JavaArray ja = (java.lang.JavaArray) d.retObj();
int[] a2 = (int[]) ja.ArrayValue;  // a2 contains .NET array { 4, 5, 6 }
```

The ArrayValue of a JavaArray may only contain arrays whose elements are primitives or objects descended from java.lang.Object. Therefore, for example, when a method declared to return Object returns an array of Strings, the proxy will return a JavaArray containing JavaStrings, as follows:

```
public class e
{
  public String[] retArray() { return new String[3] { "a", "b", "c" }; }
  public Object retObj() { return new String[3] { "d", "e", "f" }; }
}
```

.NET code would access a proxy for d as follows:

```
e o3 = new e();
string[] a = e.retArray();   // a contains .NET array { "a", "b", "c" }
java.lang.JavaArray ja = (java.lang.JavaArray) e.retObj();
java.lang.JavaString[] a2 = (java.lang.JavaString[]) ja.ArrayValue;
// a2[0].StringValue == "d"
// a2[1].StringValue == "e"
// a2[2].StringValue == "f"
```

When accessing fields declared to be of class java.lang.Object, the rules are the same as for methods that are declared to return values of java.lang.Object.

When a method expects a parameter of class java.lang.Object, one cannot pass a .NET string or array, but must wrap the string or array in a JavaString or JavaArray object. For example, if there is a Java class f as follows:

```
public class f
{
  public void passString( String s ) { }
  public void passArray( int[] i ) { }
  public Object passObj( Object o) { }
}
```

.NET code would access a proxy for f as follows:

```
f o4 = new f();
string s = "a";
int[] i = {1, 2, 3};
f.passString( s );
f.passArray( i );
f.passObj( new java.lang.JavaString( s ) );
f.passObj( new java.lang.JavaArray( i ) );
```

Unlike the case of methods returning arrays or strings nested in arrays, it is not necessary to wrap the inner strings or arrays inside JavaString or JavaArray objects; the system will automatically take care of this. Thus, the following call is legal:

```
f.passObj( new java.lang.JavaArray( { "a", "b", "c" } ) );
```

When assigning to fields declared to be of class java.lang.Object, the rules are the same as for passing strings or arrays in place of Object parameters.

Similarly, when a proxied field or method is declared to return a java.lang.Object, and an enum is returned, it is returned wrapped inside a java.lang.JavaEnum object. The actual native .NET enum can be retrieved by accessing the JavaEnum object's EnumValue property. Likewise, when a proxied method has a declared parameter of type java.lang.Object, or a field is of type java.lang.Object, and an enum is passed to the method or assigned to the field, it is necessary to wrap the enum in a JavaEnum object before passing it to the method or assigning it to the parameter.

For example, if there are Java classes g and h as follows:

```
public enum g { x, y, z };

public class h
{
  public Object passObj( Object o) { }
}
```

The proxy generator would create a native .NET enum class g and a proxy class h. .NET code would access the proxy for h as follows:

```
h o5 = new h();
java.lang.JavaEnum o6 = (java.lang.JavaEnum) h.passObj( new java.lang.JavaEnum( g.x ) );
g g1 = (g) o6.EnumValue;          // g1 is now g.x
```

Automatic implicit conversions between .NET strings and JavaStrings, between .NET arrays and JavaArrays, and between .NET enums and JavaEnums are also possible.

For example, a string can be converted to or from a JavaString in either of the following ways:

```
JavaString jas = new JavaString("foo");
jas = "foo";
string s = jas.StringValue;          // or …
s = jas;
```

An array can be converted to or from a JavaArray in either of the following ways:

```
int[] i = new int[]{1, 2, 3};
JavaArray jaa = new JavaArray(i);          // or …
jaa = i;
```

An enum can be converted to or from a JavaArray in either of the following ways:

```
JavaEnum jae = new Javaenum(g.x);     // or …
jae = g.x;                            // where g is an enum, and g.x is an enum value
```

A method f1(java.lang.Object o) in a .NET-side proxy can be passed a .NET string in either of the following ways:

```
f1(new JavaString("foo"));
f1((JavaString)"foo");
```

A method f2() declared to return values of class java.lang.Object can return strings in either of the following ways:

```
JavaString js = (JavaString) f2();
string s = js.StringValue;                    // or…
string s2 = (JavaString) f2();
```

Similarly, f1() can be passed a .NET array as follows:

```
int[] i = new int[]{1, 2, 3};
f1(new JavaArray(i));                         // or…
f1((JavaArray)i);
```

f2() can return a .NET array (in int[], in this example) as follows:

```
JavaArray ja = (JavaArray) f2();
int[] i = (int[]) ja.ArrayValue;              // or …
int[] i2 = (int[])(JavaArray) f2();
```

f1() can be passed a .NET enum (g.x, in this example) as follows:

```
f1(new JavaEnum(g.x));                        // or…
f1((JavaEnum) g.x);
```

f2() can return a .NET enum as follows:

```
JavaEnum je = (JavaEnum) f2();
g g1 = (g) ja.EnumValue;                // or …
g g2 = (g)(JavaEnum) f2();
```

Note that, when implicit/explicit conversions are applied to null values, the result is also a null value, except in the case where a null JavaEnum value is converted to an enum, in which case a NullReferenceException is thrown, because enums are value objects that cannot be null.

Note: Java enums must only be wrapped in JavaEnums in the above situations when Java enums are mapped to .NET enums. When Java enums are not mapped to .NET enums (see "Proxying Java enums," above, for information on how to set this option), they are not wrapped in JavaEnum wrappers when passed where a java.lang,Object is expected, because the proxies inherit from java.lang.Object.

Reference and value proxies

JNBridgePro supports the ability to designate proxies as "reference" or "value." A .NET reference proxy object contains a "pointer" to the underlying Java object, which continues to reside on the Java side. A .NET value proxy object contains a copy of the contents of the Java object, which may or may not continue to reside on the Java side. Reference and value proxies each have their own set of advantages.

When a reference proxy is passed as a parameter to the Java side, or returned from the Java side, only the reference is passed between .NET and Java. Since the reference is typically much smaller than the actual object, passing an object by reference is typically very fast. In addition, since the reference proxy points to a Java object that continues to exist on the Java side, if that Java object is updated, the updates are immediately accessible to the proxy object on the .NET side. The disadvantage of reference proxies is that any access to data in the underlying object (e.g., field or method accesses) requires a round trip from the .NET side to the Java side (where the information resides) and back to the .NET side.

When a value proxy is passed as a parameter to the Java side, or returned from the Java side, a copy of the object and its contents is passed between .NET and Java. Since the object itself is typically bigger than a reference, passing an object by value takes longer than passing it by reference. In addition, the value that is passed is a snapshot of the object taken at the time that it was passed. Since the passed object maintains no connection to the underlying Java object, any updates to the underlying Java object will not be reflected in the passed value proxy. The advantage of value proxies is that all data in the object is local to the .NET side, and field accesses are very fast, since they do not require a round trip to the Java side and back to get the data.

The choice of whether to use reference or value proxies depends on the desired semantics of the generated proxies as well as performance considerations. In general, one should use reference proxies, since they maintain the normal parameter-passing semantics of Java and C#. If a proxy is simply being used as a data transfer object, if there will be frequent accesses to its data, and if it is either relatively small or the frequency of accesses to data outweighs the time taken to transfer the object, make that proxy a value proxy.

Structure of value proxies

For .NET-to-Java projects, JNBridgePro supports three styles of value proxies, the *public/protected fields* style, the *Java Beans* style, and the *directly mapped* style. A public/protected fields value proxy contains copies of all the public/protected fields of the underlying Java object. A Java Beans value proxy contains public methods of the form getX() and setX() allowing access to a virtual private field X containing a snapshot of the value returned by getX() in the underlying Java object at the time the proxy was returned from the Java side. A getX() method takes no parameters and returns a value of the type of X. A setX() method takes a single parameter of the type of X, and returns no value. Java Bean-style value proxies should be used when creating proxies for Java Bean or EJB objects being returned by value. In all other cases, using public/protected fields-style value proxies is recommended. A directly mapped value proxy contains the entire contents of the underlying Java object translated into the equivalent .NET collection. For more information on directly mapped collection proxies, see "Directly-mapped collections," and "Other directly-mapped value objects," below.

Note that a value proxy object may contain members that are reference proxies. For example, if an object of class X (where X is a value proxy class) contains a field of type Y (where Y is a reference proxy class), values stored in that field will be reference proxies. Similarly, a reference proxy object may return (or take as parameters) value proxy objects.

If a Java class is to have a value proxy generated for it, it must meet several requirements. If it is to be passed from the .NET side to the Java side as a parameter (or is to be assigned to the field of a reference proxy), the Java class must have a public default constructor (that is, a public constructor taking no parameters). If it is to be used only in return values, the underlying Java class need not have a public default constructor. In addition, if the value proxy is to be of the Java Beans style and is to be passed as a parameter or assigned to a field of a reference proxy, the underlying Java class must have matched pairs of public getX() and setX() methods. If the value is only being returned from methods or field accesses, it must have the public getX() method but need not have the corresponding public setX() method. If these rules are violated, an exception will be thrown when the proxy object is improperly used.

All value proxies generated by JNBProxy have a public default constructor (regardless of whether the underlying Java class has one) which constructs an object of that class but does not initialize its data. If a value proxy explicitly constructed on the .NET side through the **new** operator, it is the

responsibility of the .NET-side program to initialize the proxy's members through field assignments or calls to setX() methods.

All Java Bean-style value proxies have public setX() methods corresponding to the public getX() methods in the underlying Java class, even if the Java class does not have such setX() methods. This is done to allow Java Bean-style value proxies to be initialized.

All value proxy classes inherit from, and implement, the same base classes and interfaces as the underlying Java object. Value proxies have methods corresponding to each public method in the underlying Java object. With the exception of static methods and of getX()/setX() methods in Java Bean-style value proxies, calling these methods will cause a System.NotImplementedException to be thrown. The reason for this is that while the methods must exist to accommodate any interfaces the proxy implements or abstract classes the proxy inherits from, it is impossible to translate the semantics of the methods of the underlying Java classes to .NET. If .NET-side equivalents of the underlying Java methods are necessary, the developer should create a new class deriving from the value proxy, and override the methods in the new class. Calls to static methods of by-value proxy classes will be passed through to the underlying Java class, as will accesses to static fields.

It is possible to designate an interface as a Java Bean-style value proxy. This causes any class also implementing that interface to become a Java Bean-style value proxy. This is particularly useful when passing Enterprise Java Beans by value, since the client code only refers to a remote interface. The actual EJBs are objects of dynamic classes implementing the remote interface; if the interface is a Java Bean-style value proxy, the returned EJB object will also be a Java Bean-style value proxy.

Consistency rules for value proxies

When value proxy classes are generated, the classes are checked against a number of consistency rules. If the rules are violated, changes may be made to the value/reference type assignments of the proxies, and the user is notified of the changes. The consistency rules (and the actions taken when they are violated) are as follows:

- Any class that inherits from a value proxy must also be a value proxy. If the derived class has been designated as a reference proxy, it is automatically changed to be a value proxy.
- If a class and its subclass are both value proxies, they must be of the same style. If they are not, the subclass's style is changed to be the same as the base class.
- An interface may be designated as a value proxy, but only if it is of the Java Bean style. If it is of the public/protected fields style, it will be changed to the Java Bean style.
- Any class implementing a Java Bean-style value interface is a Java Bean-style value proxy. If it is not, it is changed to be a Java Bean-style value proxy (unless the implementing class is a directly mapped value class, in which case it is left alone).
- Any classes nested inside a value proxy class are also value proxy classes of the same style as the enclosing class. If they are designated as reference proxies, or if they are value proxies of a different style, they are changed to be value proxies of the same style (unless the enclosing class is a directly mapped value class, in which case the nested classes are left alone).

Directly-mapped collections

In order to decrease the time required to access elements of a collection, JNBridgePro offers the ability to directly translate the contents of a Java collection to a .NET collection and vice versa. This

means that, when a Java collection object is returned from a method call, the result is translated into the equivalent .NET collection, with its contents (which may be either reference or value proxies) residing locally on the .NET side. Similarly, when a .NET collection object is passed as a parameter, the object is translated to into the equivalent Java collection, with its contents residing locally on the Java side.

By default, proxies for collection classes are generated as conventional reference proxies. To cause a Java collection class to be directly mapped to a .NET collection class, it is necessary to generate the proxy as "By value (mapped)" (see "Designating proxies as reference or value," above). Currently, the Java classes java.util.ArrayList, java.util.Vector, java.util.LinkedList, java.util.Hashtable, java.util.HashMap, and java.util.HashSet can be directly mapped.

A generated proxy for a directly mapped Java collection class has the same name as the underlying Java collection (e.g., the proxy class for java.util.Hashtable is also named java.util.Hashtable). Like other value proxies, it implements all interfaces and APIs as the underlying Java collection, and inherits from the same superclass as the underlying Java collection so that it can be passed or returned wherever an appropriate interface type or superclass type is expected. Also like other value proxies, calling a non-static method in the proxy, or any constructor other than the default constructor, causes a System.NotImplementedException to be thrown. A directly-mapped collection proxy differs from other proxies in that it has a property NativeImpl that contains the equivalent native .NET collection with a copy of the contents of the underlying Java object. For java.util.ArrayList, java.util.Vector, java.util.LinkedList, and java.util.HashSet, the NativeImpl property contains a .NET object of type System.Collections.ArrayList. For java.util.Hashtable and java.util.HashMap, NativeImpl contains a .NET object of type System.Collections.Hashtable.

Below is an example of the use of directly mapped collections. Assume that java.util.Hashtable has been generated as a directly mapped value proxy class. Also assume that c is a reference proxy object with methods f() and g(), where f() returns a java.util.Hashtable and g() takes a java.util.Hashtable as a parameter.

```
java.util.Hashtable java_hashtable = c.f();
System.Collections.Hashtable dotnet_hashtable = java_hashtable.NativeImpl;
// contents of dotnet_hashtable can now be directly accessed
. . .
System.Collections.Hashtable new_dotnet_hashtable
    = new System.Collections.Hashtable();
new_dotnet_hashtable["a"] = "1";
new_dotnet_hashtable["b"] = "2";
java.util.Hashtable new_java_table = new Hashtable();
new_java_table.NativeImpl = new_dotnet_table;
c.g(new_java_table);
```

Note that accesses of elements of a directly mapped proxy object are faster than accesses through a reference proxy object, but transferring the directly mapped object across the .NET/Java boundary takes longer than transferring the equivalent reference proxy object. It is up to the developer to weigh the benefits and disadvantages of each proxy type and to determine the appropriate type in each case.

There are a few conditions that must be observed when using directly mapped collection proxies:

- When passing a directly mapped collection object as a parameter from the .NET side to the Java side, the NativeImpl object must contain only objects that can be passed to the Java side: primitives, strings, proxy objects (including directly mapped collections containing legal objects), and arrays containing legal objects. If any other objects are contained in the collection object, a SerializationException will be thrown.

- When a directly mapped collection containing primitives is passed to the Java side, the received collection will contain "wrapped" primitives. For example, an integer 1 passed in a collection will be received as a java.lang.Integer object containing the value 1. This is because Java collections cannot contain primitives.
- Objects passed or returned inside a directly mapped collection are mapped according to JNBridgePro's mapping rules. For example, strings and arrays are mapped between native .NET and native Java versions, and if an object is returned for which a generated proxy does not exist, it is returned as the nearest class in the superclass chain for which a generated proxy exists.

Other directly-mapped value objects (dates, decimals)

For convenience, one can automatically map between the Java and .NET data types representing dates, and between the Java and .NET data types representing extended-precision values. In particular, JNBridgePro can map between Java's java.util.Date class and .NET's System.DateTime class, and between Java's java.math.BigDecimial and java.math.BigInteger classes and .NET's System.Decimal class.

By default, .NET-side proxies for Date, BigDecimal, and BigInteger are generated as conventional reference proxies. To cause one of these Java classes to be directly mapped to its corresponding .NET class, it is necessary to generate the proxy as "By value (mapped)" (see "Designating proxies as reference or value," above).

A generated proxy for a directly mapped Java class has the same name as the underlying Java class (e.g., the proxy class for java.util.Date is also named java.util.Date). Like other value proxies, it implements all interfaces and APIs as the underlying Java class, and inherits from the same superclass as the underlying Java class so that it can be passed or returned wherever an appropriate interface type or superclass type is expected. Also like other value proxies, calling a non-static method in the proxy, or any constructor other than the default constructor, causes a System.NotImplementedException to be thrown. A directly-mapped value proxy differs from other proxies in that it has a property NativeImpl that contains the equivalent native .NET collection with a copy of the contents of the underlying Java object. For java.util.Date, the NativeImpl property contains a .NET object of type System.DateTime. For java.math.BigInteger and java.math.BigDecimal, NativeImpl contains a .NET object of type System.Decimal.

Below is an example of the use of directly mapped value objects. Assume that java.util.Date has been generated as a directly mapped value proxy class. Also assume that c is a reference proxy object with methods f() and g(), where f() returns a java.util.Date and g() takes a java.util.Date as a parameter.

```
java.util.Date java_date = c.f();
System.DateTime dotnet_datetime = java_date.NativeImpl;
// contents of dotnet_datetime can now be directly accessed
. . .
System.DateTime new_dotnet_datetime
    = new System.DateTime(2004, 11, 5, 11, 04, 34); // November 5, 2004, 11:04:34AM
java.util.Date  new_java_date = new java.util.Date();
new_java_date.NativeImpl = new_dotnet_datetime;
c.g(new_java_date);
```

Note that accesses of data inside a directly mapped proxy object are faster than accesses through a reference proxy object, but transferring the directly mapped object across the .NET/Java boundary takes longer than transferring the equivalent reference proxy object. It is up to the developer to weigh the benefits and disadvantages of each proxy type and to determine the appropriate type in each case.

Also note that System.Decimal's precision, while extended, is still limited. If the value of the underlying BigDecimal or BigInteger object has a precision greater than that supported by System.Decimal, then the value of the System.Decimal returned in the proxy's NativeImpl field will be rounded to the precision supported by System.Decimal. For example, if the underlying BigDecimal object has a precision greater than 29 digits, the mapped proxy's NativeImpl field will contain a System.Decimal value that has been rounded to 29 digits.

Note: Java and .NET use differing time quanta for their Date/DateTime classes. This means that some accuracy in the microsecond/100-nanosecond range will be lost when passing .NET Dates to Java. It also means that, when passing a .NET DataTime around DateTime.MaxValue to Java and then returning the value, the returned value might be greater than DateTime.MaxValue and will cause an ArgumentOutOfRangeException to be thrown on the .NET side. Finally, it is possible to create Java Dates greater than DateTime.MaxValue; when these are passed from Java to .NET, an ArgumentOutOfRangeException will be thrown on the .NET side.

Bridge methods and ambiguous calls

When the Java compiler compiles generic classes, it sometimes automatically generates additional methods. For example, consider the following Java interface and class:

```
public interface MyInterface<T>
{
        T f();
}

public class MyClass implements MyInterface<String>
{
        String f() { return null; }
}
```

After erasure, the compiled classes resemble the following:

```
public interface MyInterface
{
        Object f();
}

public class MyClass implements MyInterface
{
        String f() { return null; }

        // compiler-generated bridge method
        Object f() { return this.f(); }          // the one above that returns String
}
```

Note the extra method, which is called a "bridge method." It is generated to ensure type fidelity, since MyInterface, after erasure, requires MyClass to implement an f() that returns Object. Bridge methods can also be generated when there are no generic classes (when covariant return types are encountered), and when there is inheritance from a class rather than an interface.

A problem arises when MyClass is proxied. The .NET proxy for MyClass contains two methods f(). If we write C# code against the proxy:

```
MyClass mc = new MyClass();
mc.f();
```

The C# compiler will emit a compile-time error reporting that there is an ambiguous call, since C# cannot tell which version of f() we are calling. In cases where we need to call f() from C#, there are ways to distinguish between the two. For example, one can assign mc to the implemented interface and call through the interface:

```
MyInterface mi = mc;
mi.f();
```

One can also use reflection to specify the method that will be called:

```
using System.Reflection;
…
MethodInfo[] methods = typeof(MyClass).getMethods();
foreach(MethodInfo method in methods)
{
        if (method.Name == "f" &&
        method.GetParameters().Length == 0 &&
        method.ReturnType == typeof(java.lang.Object))
        {
         return method.Invoke(mc, new Object[]{});
        }
}
```

Transaction-enabled classes and support for transactions

Certain Java APIs, particularly those connected with monitors or transactions, may depend on certain calls occurring in the same thread. If they do not, an exception may be thrown. For example, calls to an API to begin a transaction and to commit a transaction may need to be made from the same thread. This may make it problematic to generate proxies for such APIs and to use them from .NET. The Java side maintains a pool of threads listening for remoting calls from the .NET side. Ordinarily, the first available Java thread is chosen to execute the remoting call. In such situations, one cannot count on proxy calls from a .NET thread to be handled by the same Java thread each time.

In order to address this problem, JNBridgePro allows the user to designate proxy classes as *transaction-enabled*. All .NET calls to transaction-enabled proxy classes, even to different transaction-enabled proxy classes, from the same .NET thread are guaranteed to all be handled by the same Java thread. Conversely, .NET calls to transaction-enabled proxy classes from different .NET threads are guaranteed to be handled by different Java threads. This means, for example, that if all classes participating in a transaction have their proxies designated to be transaction-enabled, then .NET calls to those proxies can be used to drive the transaction.

It should be noted that a transaction may not be explicitly visible to the .NET caller; it may be buried inside the Java code called from .NET. In such cases, it is still necessary for the proxy classes from which the proxy operations are eventually called to be designated as transaction-enabled, if the transaction lasts across more than one .NET call. (If the duration of the transaction is a single .NET call, the proxy need not be designated transaction-enabled, since the single call is guaranteed to be handled by a single thread.) If the user calls non-transaction-enabled proxies, and a java.lang.IllegalStateException is thrown, it is likely that a transaction was attempted somewhere in the called Java code; in such cases, designating transaction-enabled proxies may solve the problem.

If an interface is designated to be transaction-enabled, then any dynamically generated proxy classes implementing that interface (for example, the implementation of an Enterprise Java Bean's remote interface) are automatically transaction-enabled. Any statically generated proxy (that is, a proxy

explicitly generated by the jnbproxy proxy generation tool) implementing a transaction-enabled interface is only transaction-enabled if the class itself is explicitly designated as transaction-enabled.

Proxy classes designated as pass-by-value cannot be transaction-enabled. If a pass-by-value proxy class is designated as transaction-enabled, that designation is ignored.

To designate a proxy class as transaction-enabled, see the section "Designating proxies as transaction-enabled," above.

Users should use the transaction-enabled feature sparingly, and only where necessary. While proxies unnecessarily designated as transaction-enabled will continue to work, transaction-enabled proxies incur a performance penalty, since the calls must pass through an additional thread management mechanism. In addition, each thread that calls a transaction-enabled proxy causes a Java thread to be reserved for it on the Java side. These threads take time to create, consume resources, and, unlike conventional threads in the Java-side thread pool, are not recycled to handle calls from multiple .NET-side threads.

Note that in earlier versions of JNBridgePro, transaction-enabled proxies were known as "thread-true." Their functionality is identical to what used to be known as "thread-true" proxies, and can be used wherever thread-true proxies were previously used.

Enabling and configuring cross-platform transactions

Java Enterprise Edition and .NET both support distributed transactions that support the X/Open standard for the two-phase commit protocol. However, because of different implementations, distributed transactions are not automatically cross-platform and transaction managers on the two platforms cannot automatically participate in global transaction processing.

JNBridgePro supports .NET-to-Java cross-platform transactions by extending an existing *implicit* transaction scope on the .NET side to a managed *explicit* transaction on the Java side. JNBridgePro will automatically create and and manage a javax.transaction.UserTransaction that is associated with the thread in which .NET-to-Java calls execute on the Java side. This user transaction is dependent on the dominant transaction scope on the calling .NET side and participates in the two-phase commit protocol managed by the Windows OS transaction coordinator.

A simple example would be a .NET-based CRM system that provides an API that creates, updates and manages customer data. In this example, a .NET transaction scope is providing data integrity for calls to the CRM system's API that manages customer data; however, it has become necessary to integrate a Java customer billing application whose API is implemented as an Entity EJB executing in a JEE application server. Using JNBridgePro to create transaction-enabled .NET proxies of the billing application's EJB remote interface, the .NET applcation can transactionally create and update customer data on both the .NET CRM and the Java billing application. On the .NET side, data integrity is ensured by the implicit transaction scope. On the Java side, data integrity is ensured by a JNBridgePro managed user transaction. If either side throws an exception during data processing, both transactions are rolled back. If no exception occurs and the two-phase commit protocol commences, if either the Java or .NET transaction managers call for a rollback during the prepare phase, both transactions are rolled back.

Enabling transactions

Enabling cross-platform transactions on the .NET side is initialized with this method

```
static void com.jnbridge.transaction.DotNetTM.enable(
```

```
System.Transactions.Transaction transaction
, int timeoutInSeconds)
```

where *timeoutInSeconds* is the timeout set for the transaction scope and *transaction* is the System.Transactions.Transaction associated with the transaction scope. This method not only ***must*** be called prior to any transaction-enabled .NET-to-Java calls, it ***must*** be called prior to any .NET calls that invoke a resource that enlists in the transaction. All .NET-to-Java proxy methods and constructors called after cross-platform transactions have been enabled must be designated transaction-enabled.

Warning: If a .NET-to-Java class that is *not* designated transaction-enabled is constructed or its methods invoked after cross-platform transactions are enabled, a JNBTransactionException will be thrown. *This will cause an immediate rollback on both platforms.*

Obtaining the .NET side transaction

In order to enable cross-platform transactions using the DotNetTM.enable() method, it is necessary to obtain the System.Transactions.Transaction object associated with the transaction scope so it can be used as the *transaction* argument to the enable() call. This can be done using the Transaction.Current static property. The following code example shows using a transaction scope and enabling cross-platform transactions.

```
using ( TransactionScope scope = new TransactionScope(
            TransactionScopeOption.RequiresNew
            , TimeSpan.FromSeconds(600) ) )
{
   try
   {
      com.jnbridge.transaction.DotNetTM.enable(Transaction.Current, 600);
    }
    catch(Exception ex)
    {
       //unable to enable cross-platform transactions
       throw new Exception("Hey! Check the configuration and javaSide.* properties", ex);
   }
   // do .NET data work
   // do .NET-to-Java data work
   scope.Complete();
}
```

Configuring the Java side for .NET-to-Java cross-platform transactions

A general use case for .NET-to-Java cross-plaform transactions is creating a .NET-to-Java proxy of the remote interface of an Entity EJB running in a JEE application server. While is is possible to proxy any Java API that enlists in transactions, the dependent javax.transaction.UserTransaction object that is created and managed by JNBridgePro can only be obtained from a JEE application server. Because of this, the Java side must be configured with properties to enable access to the the *Java Naming and Directory Interface* on the application server.

Configuration properties for the Java side can be placed in the `jnbcore.properties` file, added to the command-line that starts Java or placed in a Properties table object when calling JNBMain.start(). You can also specify these properties when using shared memory – see the note at the end of this section. For a full discussion of Java side properties see the section "Java-side properties and the

jnbcore.properties file". For more information on starting the Java side, see the section "System configuration for proxy use", above.

- **javaSide.appServerUrl**: This property is the connection URL used to connect to an application server. It is generally of the form [*protocol*]://[*hostname*]:[*port*]. As an example, the URL connection for Oracle WebLogic is `t3://scriabin.jnbridge.com:7001`.
- **javaSide.appServerContextFactory**: This property is the JNDI intial context factory class. As an example, the initial context factory for JBoss is org.jnp.interfaces.NamingContextFactory.
- **javaSide.appServerSecurityPrincipal**: The user name, if necessary. If credentials are not required, this property need not be set.
- **javaSide.appServerSecurityCredentials**: The password, if necessary. If credentials are not required, this property need not be set.
- **javaSide.appServerJNDIProperty**: This property is used for setting JEE system properties. Not all JEE application servers require properties beyond the above four, however if required, each property is numbered, e.g *javaSide.appServerJNDIProperty.1*, *javaSide.appServerJNDIProperty.2*, etc. As an example, IBM WebSphere requires this property if the SUN JRE is used in place of the IBM JRE: com.ibm.CORBA.ORBInit=com.ibm.ws.sib.client.ORB.

Note: If you are using shared-memory communications, there is no Java-side properties file, and the Java side is not started explicitly using JNBMain.start() with supplied Java-side properties. Instead, you can configure the Java-side application server settings through the jvmOptions element of the <dotNetToJavaConfig> tag (if you're using an application configuration file), or through the appropriate variant of JNBRemotingConfiguration.specifyRemotingConfiguration() (if you are configuring programmatically).

For example, if you are using an application configuration file, your <dotNetToJavaConfig> tag could have information like the following:

```
<dotNetToJavaConfig
    additional configuration information goes here,
    jvmOptions.0="-DjavaSide.appServerContextFactory=org.jnp.interfaces.NamingContextFactory"
/>
```

Coding guidelines

In order to ensure proper function of cross-platform transactions, please follow these coding guidelines. Always call com.jnbridge.transaction.DotNetTM.enable() after defining the transaction scope, but before any.NET-to-Java classes are created or their methods called. Also, call enable() before any .NET calls that may enlist in the transaction. It's good practice to catch any exceptions thrown by the enable() call.

- With the exception of the DotNetTM.enable() call, never catch exceptions thrown by the .NET-to-Java proxy calls. If an exception must be caught, either re-throw it or throw another exception in the try block. Exceptions are caught by the transaction scope notifying the transaction manager to abort the transaction and initiate a rollback before the two-phase commit commences.
- If creating a proxy of an EJB interface where the EJB is running in a JEE container with container-managed transactions, then the EJB must disable using container-managed transactions. If, not an exception will be thrown by the enable() method.
- Using explicit .NET transactions instead of a transaction scope is not permitted. System.Transactions.CommitableTransaction cannot be used in the enable() method.


JNBPRO

Using JNBridgePro for Java-to-.NET calls

This section describes how to use JNBridgePro when using it in the *Java-to-.NET* direction. Both *proxy generation* and *proxy use* scenarios are described. For information on using JNBridgePro in the *.NET-to-Java* direction, see "Using JNBridgePro for .NET-to-Java calls," and for information on *bi-directional* use, see "Use of JNBridgePro for bi-directional interoperability."

Generating proxies with JNBridgePro

Each .NET class that is to be accessible to Java classes must have a corresponding Java proxy class that manages communications with the .NET class. Java classes interact with the proxy class as though it were the underlying .NET class, and the actions are transparently communicated to the underlying .NET class. Creation of these proxy classes is typically done during application development, and is accomplished through use of the JNBProxy tool included in JNBridgePro.

JNBProxy will, given one or more .NET class names, generate Java proxies for those classes, and optionally all the supporting classes, including parameter and return value types, interfaces, and super classes, resulting in complete class closure. One can also restrict JNBProxy to generate proxies for only specified classes, and not for the entire set of supporting classes.

Note: You cannot generate a proxy for an obfuscated .NET class unless you know the obfuscated name of the class. Similarly, you cannot access an obfuscated member (a field, property, or method) of a class unless you know the obfuscated name of the member.

Use of JNBProxy results in creation of a jar (Java ARchive) containing the implementation of the Java proxy classes corresponding to the .NET classes. Once this assembly has been generated, it can be used to access the corresponding Java classes as described in the section *Using proxies with JNBridgePro*, below.

JNBProxy comes in three forms: a standalone GUI-based application, an Eclipse plug-in (for Java-to-.NET projects; for .NET-to-Java projects, there is a Visual Studio plug-in), and a command-line-based standalone application. These are described below.

JNBProxy (standalone GUI-based application)

The **JNBProxy** Windows (GUI) application (`jnbproxygui.exe`) can be used to generate Java proxies for .NET classes. (Note, only use jnbproxygui32.exe when you are running on 64-bit systems, and you are proxying classes from 32-bit DLLs. For all other situations, use jnbproxygui.exe.)

Initial invocation

The first time the JNBProxy GUI is invoked after installation, its Java options must be configured. JNBProxy displays a copy of the Java Options window, as in Figure 34 below. (Note: you may reconfigure these values later via the **Project→Java Options…** menu item, which brings up the same window.)

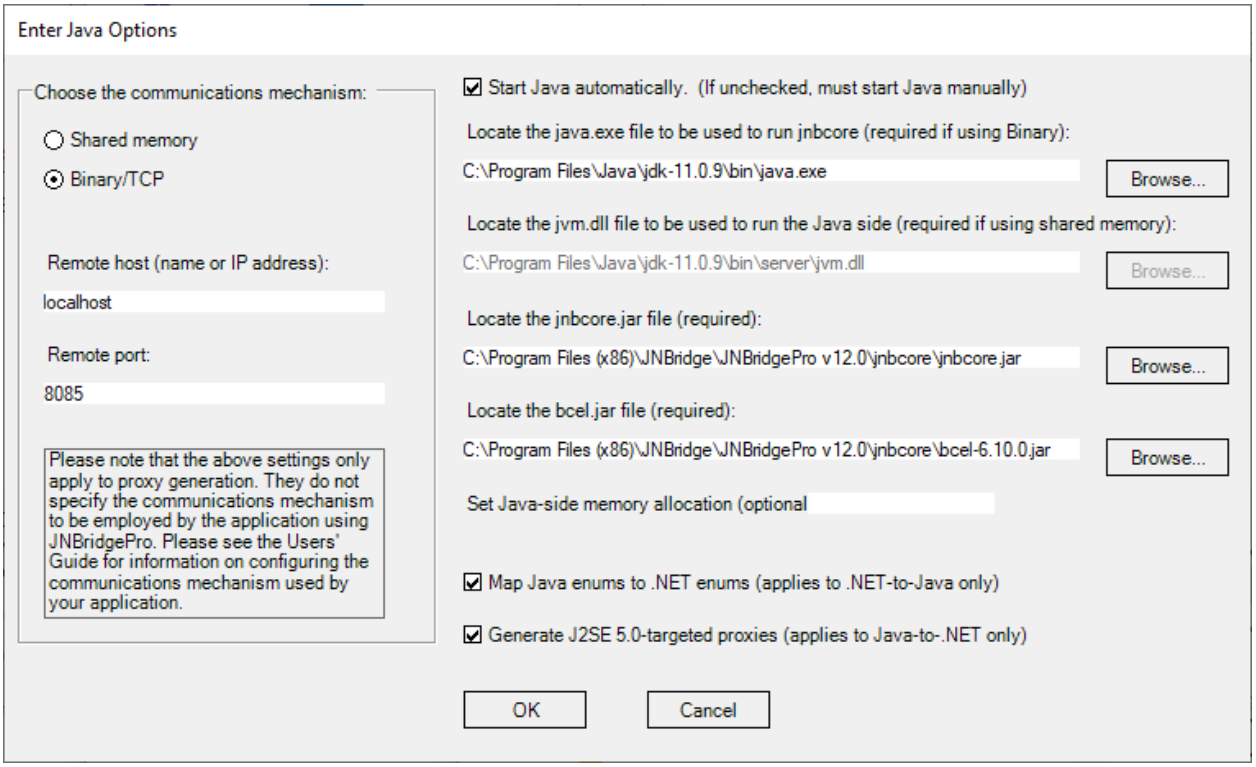


Figure 34: Java Options window

JNBProxy will attempt to fill in these options itself. Typically, there is no need to do anything further; the supplied choices will be sufficient. However, if JNBProxy cannot find a necessary file, or if unusual conditions apply, you may need to explicitly specify one or more of the options.

Two communications mechanisms are available for managing the communications between the .NET and Java sides: shared memory, and binary/TCP. If you select binary/TCP, you must specify the host on which the Java side resides, and the port through which it will listen to requests from the .NET side. If you select shared memory, you must locate the file jvm.dll that implements the Java Virtual Machine that will be run in-process with the .NET-side code. Typically, jvm.dll is included as part of

the JDK (Java Development Kit) or JRE (Java Runtime Environment) on which you will be running the Java side.

The default settings (binary/TCP, localhost, and port 8085) are typically sufficient and usually need not be changed.

Note: While shared memory is faster than binary/TCP, it has not been chosen as the default setting due to certain limitations in the Java Native Interface (JNI), which require that JNBProxy be exited and restarted whenever JNBProxy's classpath is changed. If this is not an issue, using shared memory with JNBProxy is recommended.

If Java is to be started automatically, you must supply the location of the Java runtime (`java.exe`), along with the locations of the `jnbcore.jar`, `jnbcore.properties`, and BCEL jar files.

If you wish to start Java manually, uncheck the "Start Java automatically" box.

Note: if the Java side is located on another machine, JNBProxy cannot start it automatically; it must be started manually.

When JNBProxy is launched, the application displays a *launch form* (Figure 35). Depending on which radio button is selected, the user can create a new .NET-to-Java project, a new Java-to-.NET project, a recently-used project, or some other existing project file. After the project is selected, the main form of JNBProxy will appear (Figure 36).

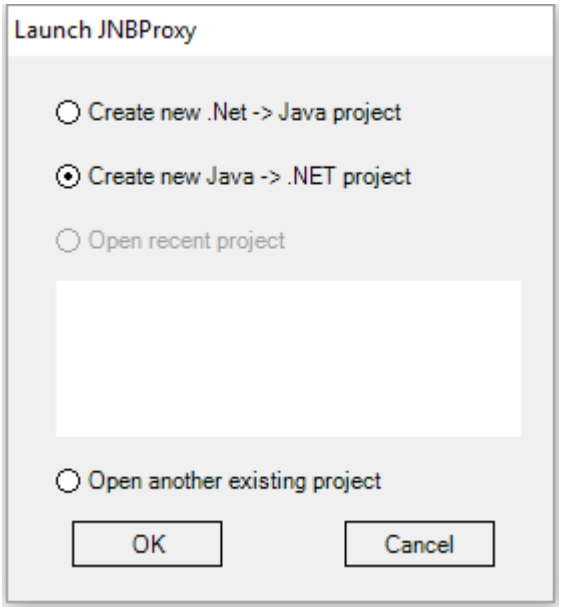


Figure 35. JNBProxy launch form

GUI layout

Figure 36 shows the GUI version of the JNBProxy tool immediately after it is launched. Note that the title bar shows the direction of the current project, and indicates which type of proxies (.NET or Java) will be created. In Figure 36, the title bar shows that the current project is a Java-to-.NET project, and that Java proxy classes for underlying .NET classes will be generated. Within the tool's window are four panes: the *environment pane*, the *exposed proxies pane*, the *signature pane*, and the *output pane*.

The environment pane displays the .NET classes of which jnbproxy is aware, and for which proxies can be generated. The exposed proxy pane shows those .NET classes for which proxies will be

generated. The signature pane shows information on the methods and fields offered by a Java class, as well as other information on the class. The output pane displays diagnostic and informative messages generated during the operation of the tool. You may drag the splitter controls to resize the panes.

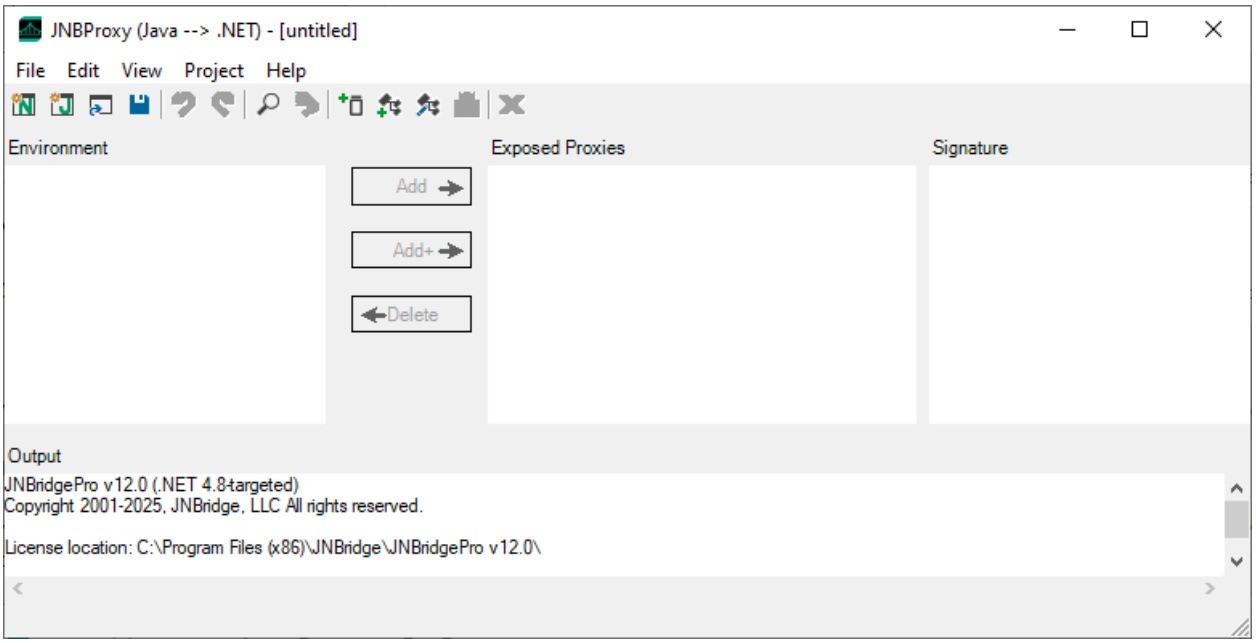


Figure 36. JNBProxy proxy generation tool (GUI version)

The title bar of JNBProxy contains an indication of whether the current project is Java-to-.NET or .NET-to-Java. In Figure 36, we see that the current project is Java-to-.NET.

Process for generating proxies

Generating proxies takes three steps:

- Add classes to the environment for which you might want to generate proxies.
- Select classes in the environment for which proxies are to be generated and add them to the exposed proxies list.
- Build the proxies into a Java jar file.

Adding classes to the environment

The first step in generating proxies is to load candidate classes into the environment. Think of the environment as a palette from which one can choose the proxies that are actually generated. Classes can be loaded into the environment in two ways: from a .NET assembly DLL or EXE file, and as specific classes found among a set of assemblies.

To load classes from a .NET assembly, click on the menu item **Project→Add classes from assembly file...**. A dialog box will appear allowing the user to locate one or more assembly files whose classes will be loaded. Opening an assembly file will cause all the classes inside of it to be added to the environment pane. The progress of the operation is shown in the output pane. The operation can be terminated before completion by clicking on the Stop button located on the GUI's tool bar.

Note: any assembly file from which classes are to be loaded must be in the assembly list. To edit the assembly list, see "Setting or modifying the assembly list," below.

Note: The results of operations terminated prematurely by clicking the stop button may be inconsistent and should not be used. These partial results can be removed by performing an Undo operation.

To load specific classes from the assembly list, click on the menu item **Project→Add classes from assembly list…**. When this item is selected, the Add Classes dialog box is displayed (Figure 37).

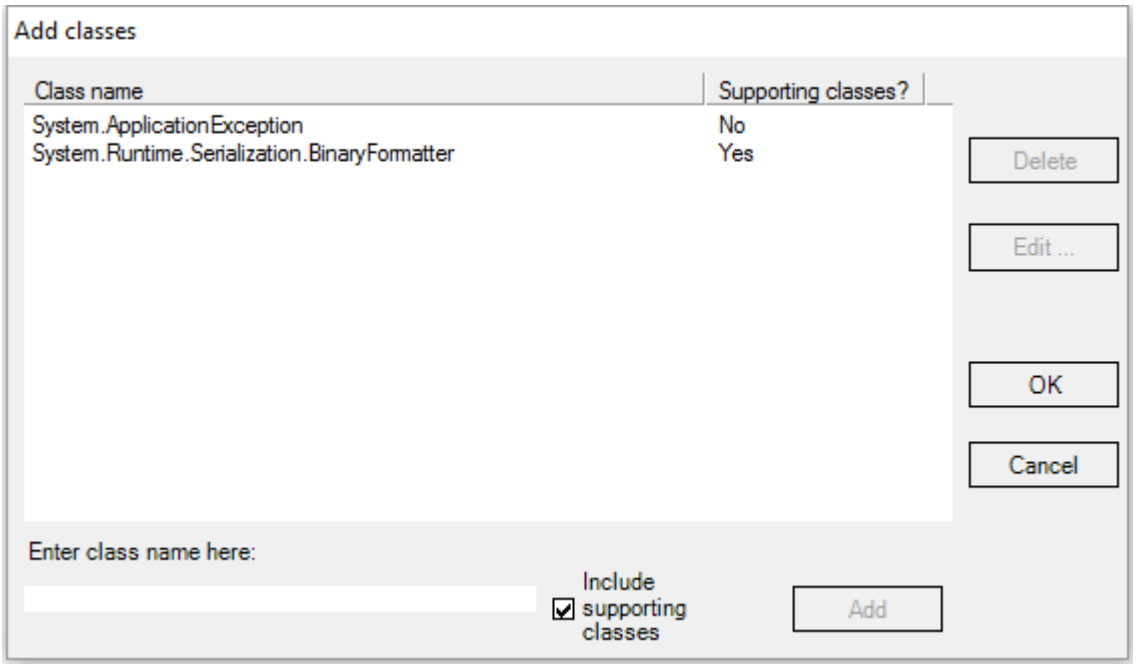


Figure 37. Add classes dialog box

To specify a class to be added, type the fully-qualified class name into the text box at the bottom of the dialog box. (Note that, as you type, the interface will provide class name suggestions based on what is available and what you have typed so far.) Leave checked the "Include supporting classes" check box to indicate that all classes supporting the specified class should be automatically added. (See *Supporting classes*, below, for more information on supporting classes.) Then, click the Add button to add the class to the list of classes to be loaded into the environment. You may delete a class from this list by selecting the class in the list and clicking on the Delete button. You may edit the information specified for the class by selecting the class and clicking on the Edit button or by double-clicking on the class. When you are done specifying classes, click the OK button to add the classes to the environment. This process may take a few minutes, especially if any of the classes in the list must also have their supporting classes loaded. The operation can be terminated before completion by clicking on the Stop button located on the GUI's tool bar.

To load a generic class into the environment, enter the class name using the format *className`numParmams*, where *className* is the name of the class without the generic parameters, *numParams* is the number of generic parameters, and where *className* and *numParams* are separated by a backquote (`). For example, to load System.Collections.Generic.List<T>, use the class name System.Collections.Generic.List`1. (Note that the name System.Collections.Generic.List<> will also be accepted, although we recommend using the backquote format.)

Also note that, in addition to specifying the fully-qualified name of a class to be added, you can also use a trailing '*' as a wildcard, to indicate that all classes in the assembly list that match begin with the string that precedes the asterisk should be added. For example, supplying 'System.Collections.*' indicates that all the classes in the System.Collections.* package should be added.

Note: If you are doing Java-.NET co-development, and a .NET class is changed after the class has been loaded into the environment (e.g., a method or field has been added or removed, or the signature changed), the class can be updated in the environment simply by loading it again.

Note: Any class or supporting class that is added must be in the assembly list or a member of the mscorlib.dll assembly.

Note: Proxies System.Object and System.Type are always generated and added to the environment even if they are not explicitly requested or implicitly requested by checking "Include supporting classes."

Note: The results of operations terminated prematurely by clicking the stop button may be inconsistent and should not be used. These partial results can be removed by performing an Undo operation.

Selecting proxies to be generated

Once classes have been loaded into the environment, they are displayed in the environment pane, which is a tree view listing all the classes loaded into the environment grouped by package. Next to each item in the tree view is an icon indicating whether the item is a package, an interface, and exception, or some other class. (See Figure 38).

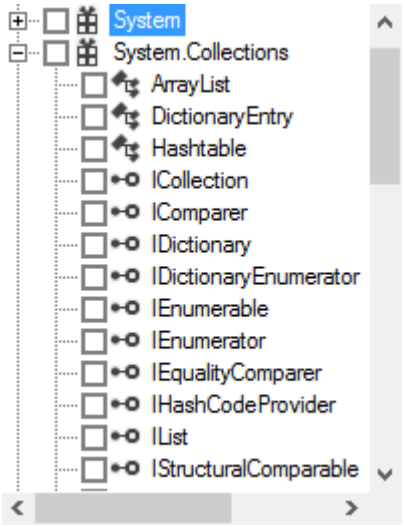


Figure 38. Environment pane

To select a class for proxy generation, check the class's entry in the environment pane by clicking on the check box next to the name. To select all the classes in a package, check the package's entry by clicking on its check box. Once a set of classes has been checked, add them to the set of exposed proxies by clicking on the **Add** button.

Alternatively, you may click on the **Add+** button to add the checked classes and all supporting classes. For a discussion on supporting classes, see the *Supporting Classes* section later in this guide. The checked classes and all supporting classes (if **Add+** was clicked) will appear in the exposed proxies pane, which is a tree view similar to the environment pane.

Note: use of* Add+ *may take a few minutes for the operation to complete. The operation can be terminated before completion by clicking on the Stop button located on the GUI's tool bar. The results of operations terminated prematurely by clicking the stop button may be inconsistent and should not be used. These partial results can be removed by performing an Undo operation.

To selectively remove classes from the exposed proxies pane so that proxies are not generated, check the entries in the exposed proxies pane for all classes or packages to be removed (by clicking on the check boxes next to the class or package names), then click on the **Delete** button. The checked items will be removed from the exposed proxies pane.

As a convenience, the Edit menu contains items to check or uncheck all the items in the environment or exposed proxy panes.

Add, **Add+**, and **Delete** operations may be repeatedly performed until the exact set of proxies to be generated appears in the exposed proxies pane.

The most recent **Add**, **Add+**, and **Delete** operations may be undone and then redone using the **Edit→Undo** and **Edit→Redo** menu items.

Note: Proxies for System.Object and System.Type will always be added to the exposed proxies pane when the Add button is clicked, even if they have not been checked in the environment pane. Also, System.Object and System.Type cannot be checked in the exposed proxies pane, and therefore cannot be deleted from that pane. The reason for this behavior is to ensure that proxies for Object and Type are always generated.

Designating proxies as reference or value

Before generating proxies for the classes listed in the exposed proxies pane, the user can designate which proxies should be reference and which should be value. (See the section "Reference and value proxies," below, for more information on the distinction between reference and value proxies.) The default proxy type is reference; if the user wants all proxies to be reference, nothing additional need be done.

To set a proxy's type (i.e., to reference or either of the two styles of value), position the cursor over the proxy class to be changed (in the exposed proxies pane), and right-click on the class. A pop-up menu will appear. Choose the desired proxy type. After the proxy type is selected, the proxy class will be color coded to indicate its type (black for reference, blue for value (public/protected fields/properties style), or green for by-value (mapped).

To set the types of multiple proxy classes at the same time, click on the **Project→Pass by Reference / Value…** menu item. The Pass by Reference / Value dialog box is displayed (Figure 39). All proxy classes in the exposed proxies pane are listed in this dialog box. To change the types of one or more classes, select those classes (left-click on a class to select it, and shift-left-click or ctrl-left-click to do multi-selects), then click on the Reference, Value (Public/protected fields/properties), or Value (Mapped) button to associate the selected classes to the desired type. When done, click on the OK button to actually set the classes to the chosen types and dismiss the dialog box, click on the Apply button to set the classes without dismissing the dialog box, or click on the Cancel button to discard all changes.

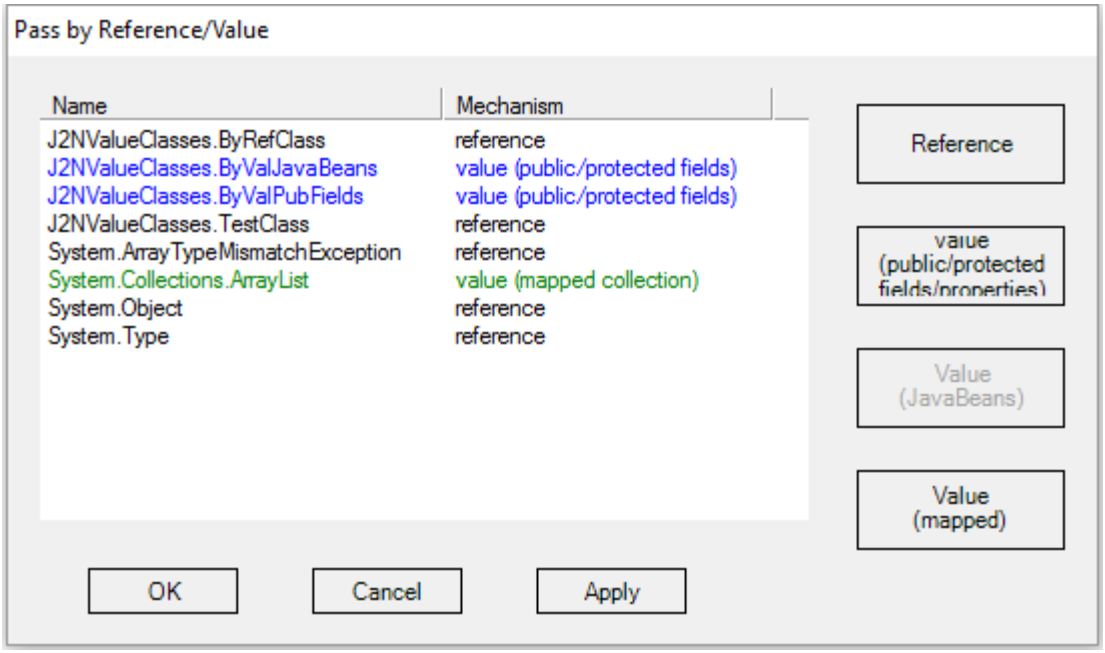


Figure 39. Pass by Reference/Value dialog box

Designating proxies as transaction-enabled (formerly "thread-true")

Before generating proxies for the classes listed in the exposed proxies pane, the user can designate which proxies should be *transaction-enabled*. (See the section "Transaction-enabled classes and support for transactions," below, for more information on transaction-enabled classes.) The default proxy type is not transaction-enabled; if the user wants all proxies to be non-transaction-enabled, nothing additional need be done.

To set a proxy as transaction-enabled, position the cursor over the proxy class to be changed (in the exposed proxies pane), and right-click on the class. A pop-up menu will appear. Select the "transaction enabled" menu item so that it is checked. To remove the transaction-enabled property from a proxy class, perform the same operation to uncheck the transaction-enabled menu item. It is currently not possible to make multiple classes transaction-enabled at the same time.

Users should use the transaction-enabled property sparingly, as it incurs a performance penalty. See the section "Transaction-enabled classes and support for transactions" for more information.

Generating the proxies

Once classes have been selected for proxy generation and have been moved into the exposed proxies pane, a Java jar file with the proxies can be generated by clicking on the **Project→Build...** menu item. A dialog box will be displayed that allows the user to specify the location and name of the jar file that will contain the generated proxies.

Note: The Build operation may take a few minutes for the operation to complete. The operation can be terminated before completion by clicking on the Stop button located on the GUI's tool bar. The results of operations terminated prematurely by clicking the stop button may be inconsistent and should not be used.

Saving and restoring projects

Work done during a JNBProxy project session can be saved in a JNBProxy project file. A project file contains the current Java classpath and the current contents of both the environment and exposed proxies panes. A project file is loaded by clicking on the menu item **File→Open Project…**, which opens a dialog that allows the user to identify the project file to be opened. If the current project has been modified, the user is offered the opportunity to save it.

Alternatively, the **File→Recent Projects** menu item allows access to the four most recently used projects.

A project is saved by clicking on the menu item **File→Save Project** or **File→Save Project As…**. The latter option lets the user specify the name and location of the project file, while the former option saves the session under its current name if it has one; otherwise it asks the user to specify the name and location of the new project file.

A new project is created by clicking on the menu item **File→New .NET->Java Project…** or **File→New Java->.NET Project…**.depending on the desired direction of the new project. If the current project has been modified, the user is offered the opportunity to save it.

Exporting a class list

Selecting the **File→Export classlist…** menu item will cause a text file to be generated that lists the classes in the Exposed Proxies pane, plus information on whether the classes are by-reference or by-value, and whether the classes are transaction-enabled. The exported class list is in the correct format to be used as input with the '/f' option of the command-line version of JNBProxy. (See the section "Using JNBProxy (command-line version)").

Generating a command-line script

In certain cases, it will be useful or necessary to incorporate proxy generation into an automated build process. The command-line version of the proxy generator (see below) is intended for these situations. To simplify the creation of command-line scripts, JNBProxy can be used to automatically generate a script for the current project. Select the **JNBridgePro→Generate command-line script…** menu item to generate a .bat file that can be used to invoke the command-line proxy generator. When generating a command-line script, the user will be prompted to supply the name of the .bat file. At the same time, a class list will be generated (see "Exporting a class list," above). If the .bat file is named *myFile.bat*, the class list will be named *myFile_classlist.txt*, unless a file of that name already exists, in which case the user will be asked whether it should be overwritten, or whether the file should have a different name.

Note that the .bat file can, and in many cases should, be edited, particularly if it will be run in a different location from that to which it was written. For information on the command-line options and how they can be edited, please see the section "Using JNBProxy (command-line version)," below.

Examining class signatures

JNBProxy displays information about a Java class so that a user can determine whether it contains properties of interest to the user. If a class item in either the environment or exposed proxies pane is selected, its information is displayed in the signatures pane. The information includes the class's name, superclass, attributes, interfaces, fields, properties, and method signatures (Figure 40).

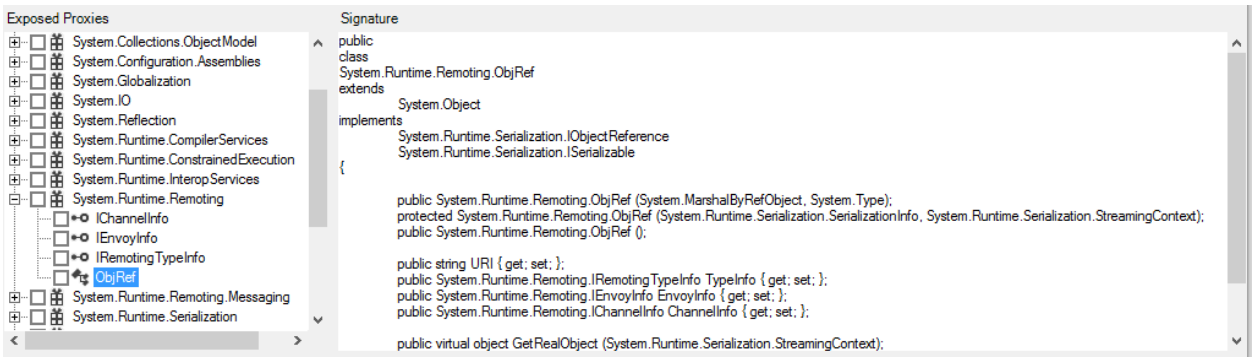


Figure 40. Selected item in exposed proxy pane and corresponding information in signature pane

Searching for classes in the environment or exposed proxy panes

To find a class among a large number of classes in the environment or exposed proxy panes, use the Find facility available by clicking on the **Edit→Find** menu item, which displays a Find window (Figure 41). The Find window offers a variety of options, including the ability to search the environment or exposed proxy panes, to search down or up, to look for exact whole-word matches, and to perform case-dependent matches. To repeat a search, the user should use the **Edit→Find Next (F3)** menu item.

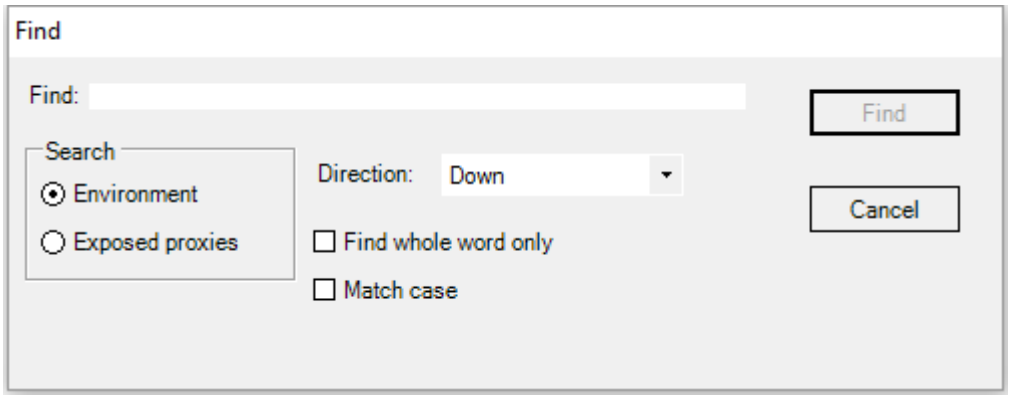


Figure 41. Find window.

Modifying tree view properties

To change the ways that the environment and exposed proxy panes are displayed, click on the **View→Tree Options...** menu item. When this option is selected, a dialog box is displayed that allows the user, for either the environment or exposed proxy pane, to show or hide the icons in the pane, and to completely expand or completely collapse the tree view in the pane (Figure 42).

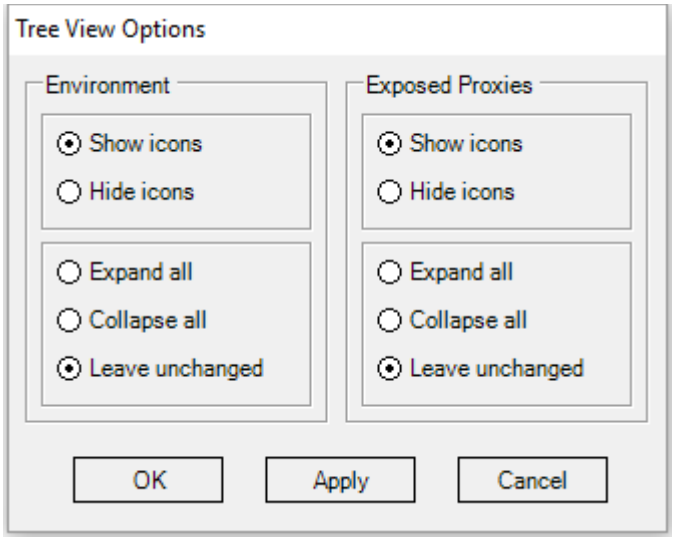


Figure 42. Tree options window

Refreshing the display

To refresh the contents of the four panes (environment, exposed proxy, signature, and output) in the JNBProxy window, select the **View→Refresh (F5)** menu item.

Setting Java startup options

The user may choose for JNBProxy to automatically start the Java side, or the user may start it manually. (See *Starting Java manually* for more information.) To control whether or not Java is started automatically, and to specify the location of various executables and libraries, select the **Project→Java Options…** menu item (Figure 43).

Two communications mechanisms are available for managing the communications between the .NET and Java sides: shared memory, and binary/TCP. If you select binary/TCP, you must specify the host on which the Java side resides, and the port through which it will listen to requests from the .NET side. If you select shared memory, you must locate the file jvm.dll that implements the Java Virtual Machine that will be run in-process with the .NET-side code. Typically, jvm.dll is included as part of the JDK (Java Development Kit) or JRE (Java Runtime Environment) on which you will be running the Java side.

The default settings (binary/TCP, localhost, and port 8085) are typically sufficient and usually need not be changed.

Note: While shared memory is faster than binary/TCP, it has not been chosen as the default setting due to certain limitations in the Java Native Interface (JNI), which require that JNBProxy be exited and restarted whenever JNBProxy's classpath is changed. If this is not an issue, using shared memory with JNBProxy is recommended.

To cause Java to be started automatically by JNBProxy, check the check box at the top of the Java options window. Leave the box unchecked if you wish to start Java manually.

If Java is to be started automatically, you must supply the location of the Java runtime (`java.exe`), along with the locations of the `jnbcore.jar` and `jnbcore.properties` and the BCEL jar files.

(the BCEL jar file is only used in the Java-to-.NET direction, but its location must always be specified.)

If you wish to start Java manually, uncheck the "Start Java automatically" box.

Note: if the Java side is located on another machine, JNBProxy cannot start it automatically; it must be started manually.

If the user's Windows account contains privileges allowing it to write new settings to the registry, the Java Options settings will be saved from one invocation of JNBProxy to the next; if the account does not have such privileges, the settings will not be saved.

Adjusting the Java-side memory allocation during proxy generation

When generating proxies for a very large number of proxies (for example, all the classes in a very large jar file), it is possible that the Java side may run out of memory, and proxy generation will fail. When this happens, one can avoid the problem by specifying a larger Java-side memory allocation to be used during proxy generation.

To adjust the Java-side memory allocation, bring up the Java Options dialog box by selecting the **Project→Java Options…** menu item. The Java Options window (Figure 43) will be displayed. One can specify a new Java-side memory allocation by entering it in the box labeled "Set Java-side memory allocation." The value entered there is the new Java-side memory allocation in bytes, and may be either a number that is a multiple of 1024, a number followed by "m" or "M" (denoting megabytes), or a number followed by "k" or "K" (denoting kilobytes). If the entry is left blank, the default memory allocation for the current Java runtime environment will be used.

Note: The Java-side memory allocation setting only applies to proxy generation. It does not affect subsequent use of the proxies. To specify a Java-side memory allocation to be used during the running application, you need to supply a –Xmx option to the Java side.

Note: If the "Start Java automatically" checkbox is unchecked, the Java-side memory allocation option will be ignored.

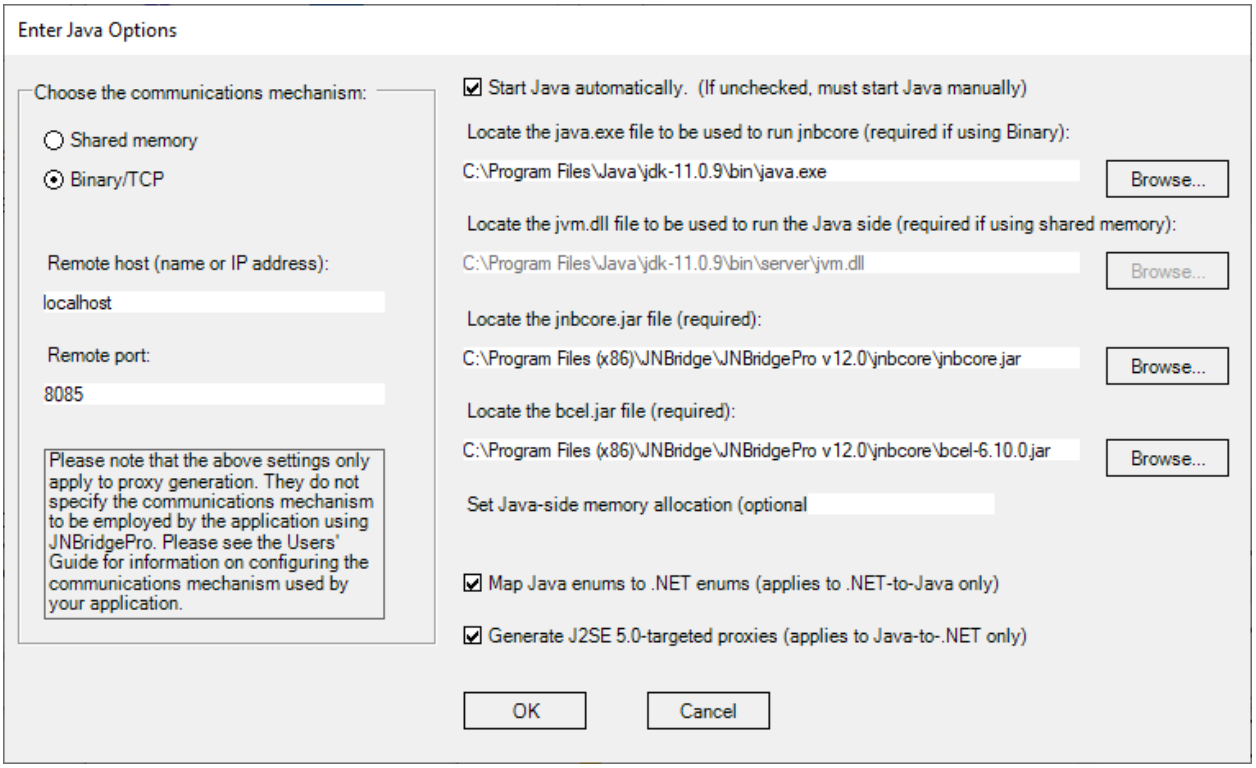


Figure 43. Java Options window.

Generating Java SE 5.0-targeted proxies

By default, generated Java-side proxies are compatible with Java SE 5.0 and later, and can therefore take advantage of mappings from .NET features such as generics, enums and vararg methods, to the equivalent Java features, all of which were introduced with Java SE 5.0. If you wish to target the generated proxies to JDK 1.1 (and therefore make them compatible with 1.1 through 1.4), you must explicitly turn off this option.

To make sure that proxies are targeted toward Java SE 5.0, bring up the Java Options window (Figure 43) by selecting Project→Java Options… and check the "Generate Java SE 5.0-targeted proxies" check box. This box is checked by default. To target the proxies to earlier versions of Java, uncheck the box.

Note: Java-side proxies targeted to Java SE 5.0 will not work with earlier versions of Java. However, you can use earlier versions of Java to generate Java SE 5.0-targeted proxies.

Setting or modifying the assembly search path

Before loading .NET types for a Java-to-.NET project, the user needs to specify the assemblies from which these types will be loaded. The **Project→Edit Assembly List…** menu item brings up an **Edit Assembly List** dialog box (Figure 44), similar to the **Edit Classpath** dialog box in .NET-to-Java projects. Added assemblies can be either EXE (executable) or DLL files, but they must be .NET assemblies. If left empty, classes can only be loaded from the standard mscorlib.dll assembly.

Assemblies can be added to the assembly list from specific file locations, or they can be added to the assembly list from the Global Assembly Cache (GAC). To add an assembly from the GAC, click the **Add from GAC…** button in the **Edit Assembly List** dialog box. A **Select Assemblies from GAC…** dialog box will be displayed (Figure 45). Simply select the assemblies to be added (one can select multiple assemblies by shift-clicking or ctrl-clicking on the assemblies in the displayed list), then click on the OK button to add the selected assemblies to the assembly list. Note that different versions of the same assembly can be in the GAC and will be displayed in the list. The user should be careful to select the desired version when adding assemblies from the GAC.

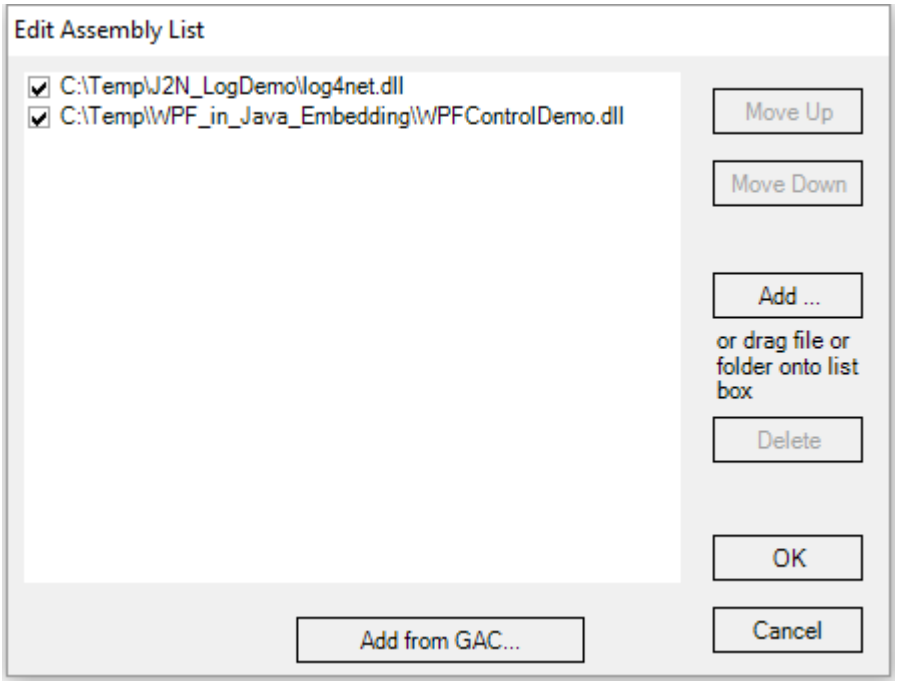


Figure 44. Edit Assembly List dialog box.

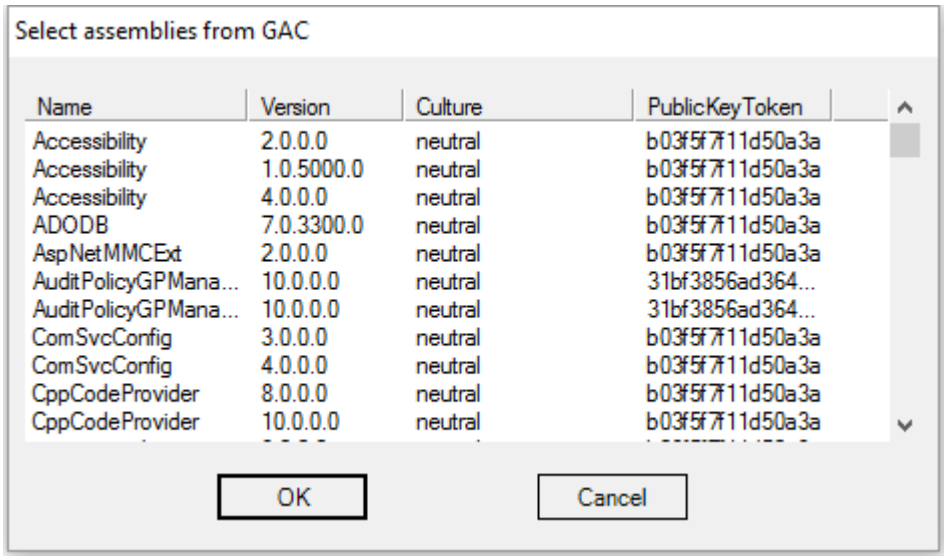


Figure 45. Select Assemblies from GAC dialog box

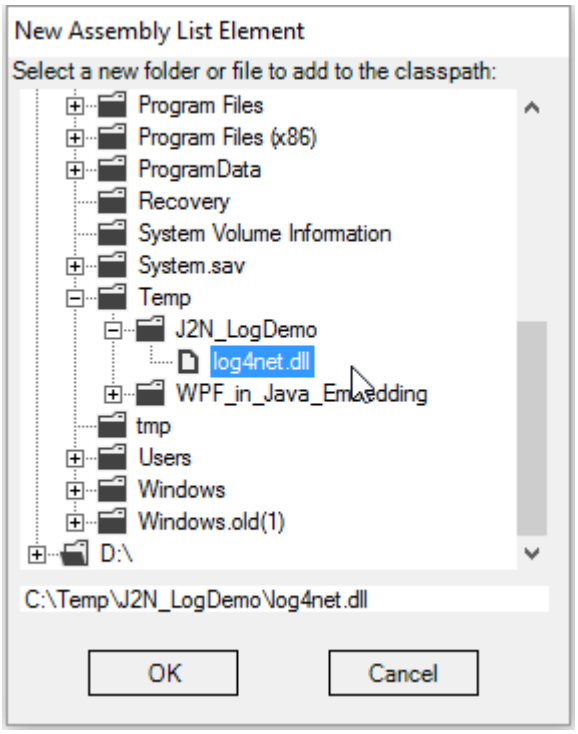


Figure 46. New Assembly List Element window.

The assembly list can be rearranged by selecting an assembly element and clicking on the **Move Up** or **Move Down** buttons, or by selecting an element and clicking on the **Delete** button. Note that only checked elements will be added to the assembly list when the dialog is dismissed.

If a type appears in more than one assembly in the assembly list, then its definition will be taken from the first assembly in the list in which the type is defined.

To add a folder or jar file to the classpath, simply drag and drop one or more files or folders onto the Edit Classpath form. You can also add a folder or jar file by clicking on the **Add...** button. This causes a New Assembly List Element window to be displayed. In this window, the user can navigate to the desired folders or assembly files, or can enter a file path directly (Figure 46). The New Assembly List Element window supports multiple selection - multiple folders and/or assembly files may be selected by ctrl-clicking, while a range of folders and/or assembly files may be selected by shift-clicking. Clicking on the **OK** button will cause the indicated folders or files to be added to the Edit Assembly List window. Selecting a folder will cause all the assemblies in the folder to be added to the assembly list.

Note: You can also type a directory or file path directly into the New Assembly List Element dialog box. This path can be an absolute path, or a UNC path to a shared network folder (e.g., \\MachineName\FolderName\FileName.dll).

JNBProxy (Eclipse plug-in)

The **JNBProxy** plug-in for Eclipse can be used to generate Java proxies for .NET classes. It works with Eclipse 3.2 through 4.13.

System requirements and plug-in installation

Before installing the plug-in, make sure you have installed any version of Eclipse, from 3.2 through 4.13. Also make sure you have installed .NET Framework 2.0, 3.0, 3.5, 4.0, 4.5, 4.6, 4.7, or 4.8.

To install the plug-in, locate the file jnbridgepro11_0_0-eclipse.zip in the JNBridgePro installation folder. Extract the contents of this zip file (a folder named com.jnbridge.plugin.eclipse_2.4.0) into the plugins folder of your Eclipse installation.

If you are using the 64-bit version of Eclipse, you must use the 64-bit version of JNBridgePro, and you must add the following argument to eclipse.ini:

```
-vm C:/Program Files/Java/jre8/bin/javaw.exe
```

or use a path to some other 64-bit javaw.exe. *Do not* use the path to the javaw.exe that resides in \Windows\System32.

Initial setup

Before using the Eclipse plug-in, configure it by selecting the **Window→Preferences…** menu item. Select the JNBridgePro section, then select the Java Options tab (Figure 47).

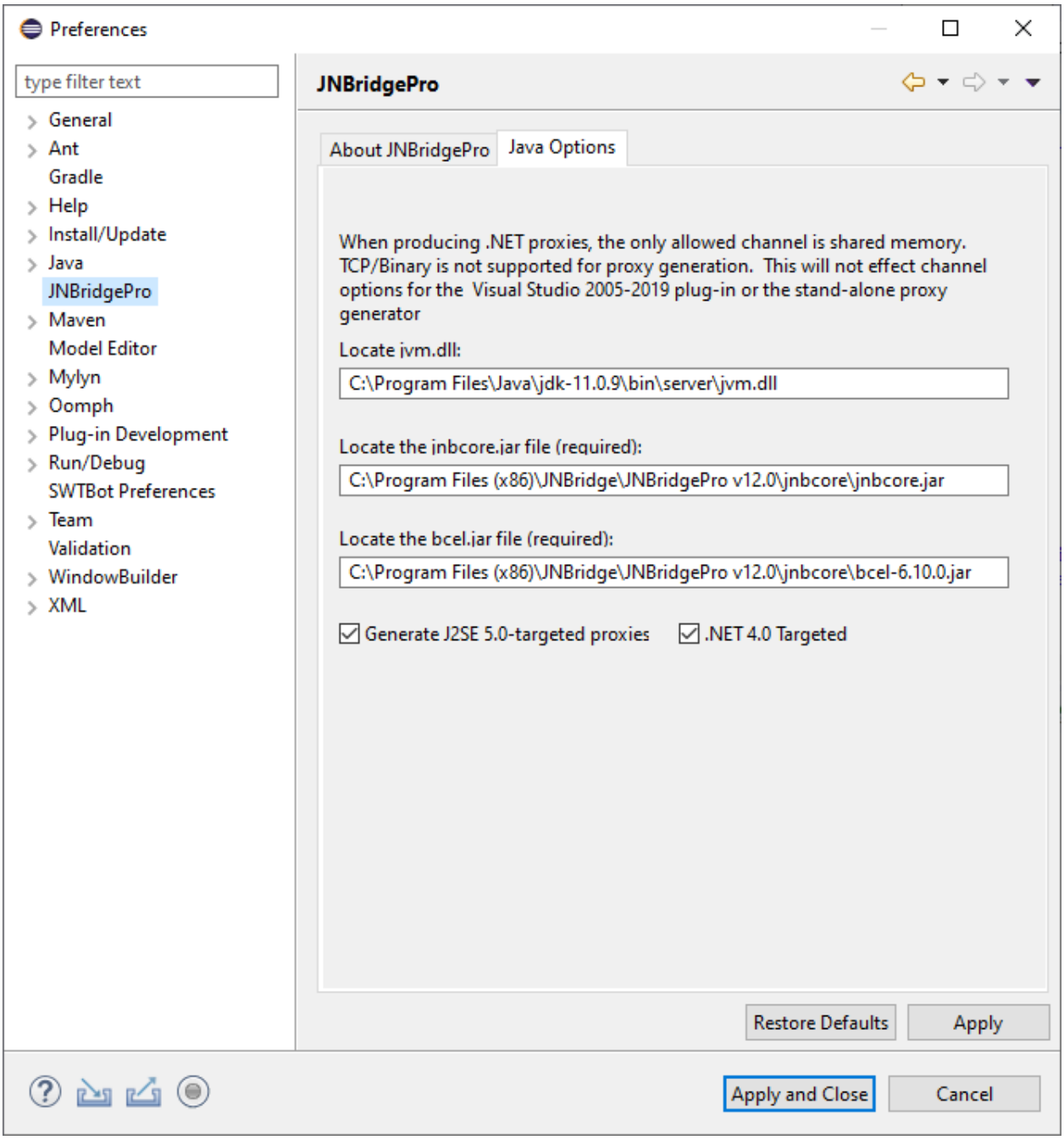


Figure 47: Java Options window

JNBProxy will attempt to fill in these options itself. Typically, there is no need to do anything further; the supplied choices will be sufficient. However, if JNBProxy cannot find a necessary file, or if unusual conditions apply, you may need to explicitly specify one or more of the options.

You must supply the paths to the files jnbcore.jar and bcel-*6.n.m*.jar (found in the jnbcore folder of the JNBridgePro 12.0 installation), and jvm.dll (located in your JDK or JRE installation). You may

enter the paths to these files by double-clicking on the text boxes, then navigating to the appropriate file, or by typing in the path to the file.

Creating a new JNBridge project

To create a new JNBridge project, select the File→New→Other… menu item. The New dialog box will be displayed (Figure 48). In the JNBridge section, select "Java to .NET Interoperability Project." Proceed through the wizard, supplying the name and location of the new project. The new project will be displayed in the Eclipse package explorer.

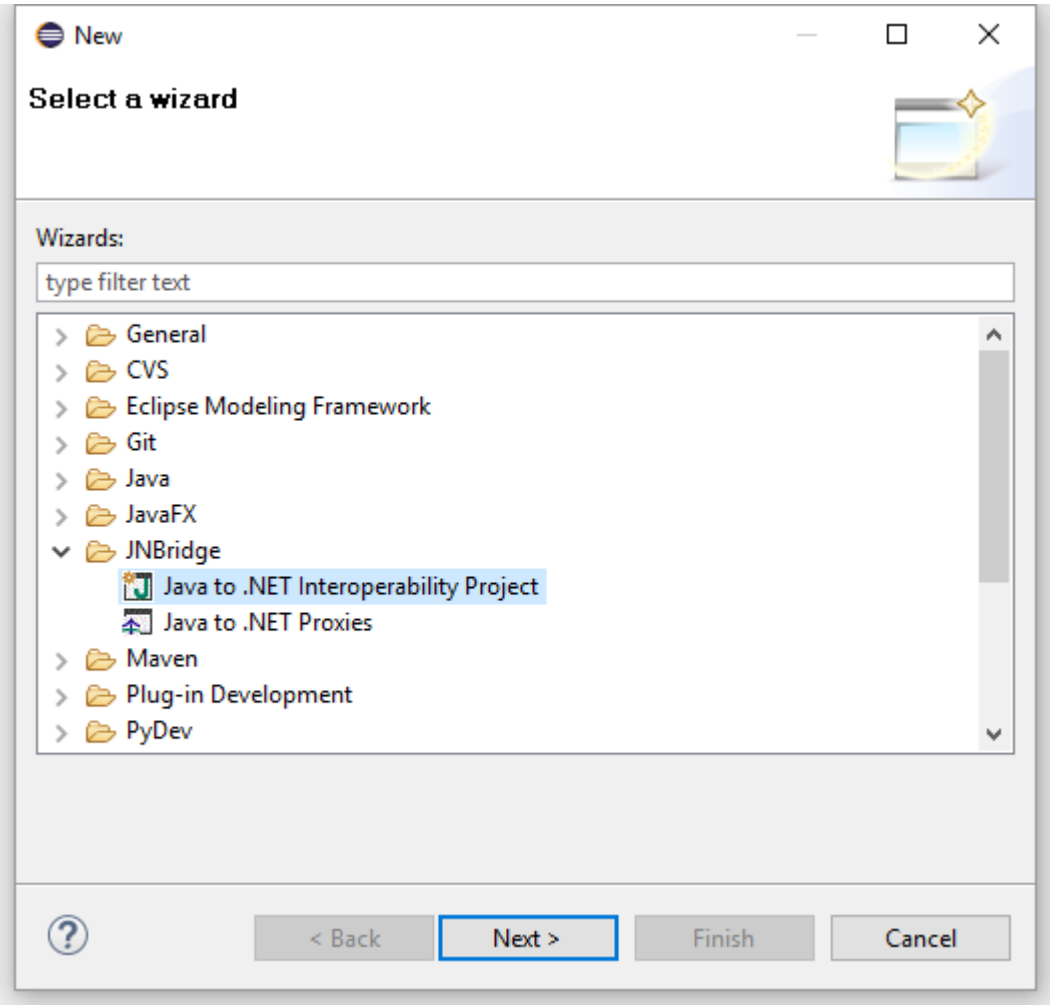


Figure 48. New dialog box

After creating the new project, create the proxy generation document by right-clicking on the newly created project node in the package explorer, and selecting **New→Other…**. The New dialog box (Figure 48) will be displayed. In the JNBridge section, select "Java to .NET Proxies." Work through the wizard, specifying the name and parent project of the proxy generation document. A new proxy generation document node will be created in the package explorer beneath the newly created project node (Figure 49). This new document node represents the .jnb file that contains all the information

needed to generate proxies. It can also be read and edited by the standalone JNBProxy GUI-based proxy generator.

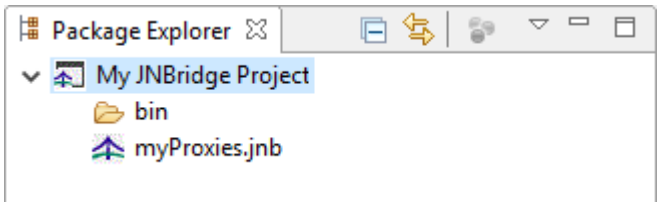


Figure 49. Package Explorer after creating new JNBridgePro project and document

GUI layout

By double-clicking on a proxy generation document (.jnb) node in the package explorer, you can open the document in an editor. Figure 50 shows the JNBProxy editor immediately after it is launched. Within the tool's window are three panes: the *environment pane*, the *exposed proxies pane*, and the *signature pane*. There is also a console window that is associated with JNBridgePro; it can be viewed by selecting the **Window→Show View→Console** menu item.

The environment pane displays the .NET classes of which jnbproxy is aware, and for which proxies can be generated. The exposed proxy pane shows those .NET classes for which proxies will be generated. The signature pane shows information on the methods and fields offered by a Java class, as well as other information on the class. You may drag the splitter controls to resize the panes.

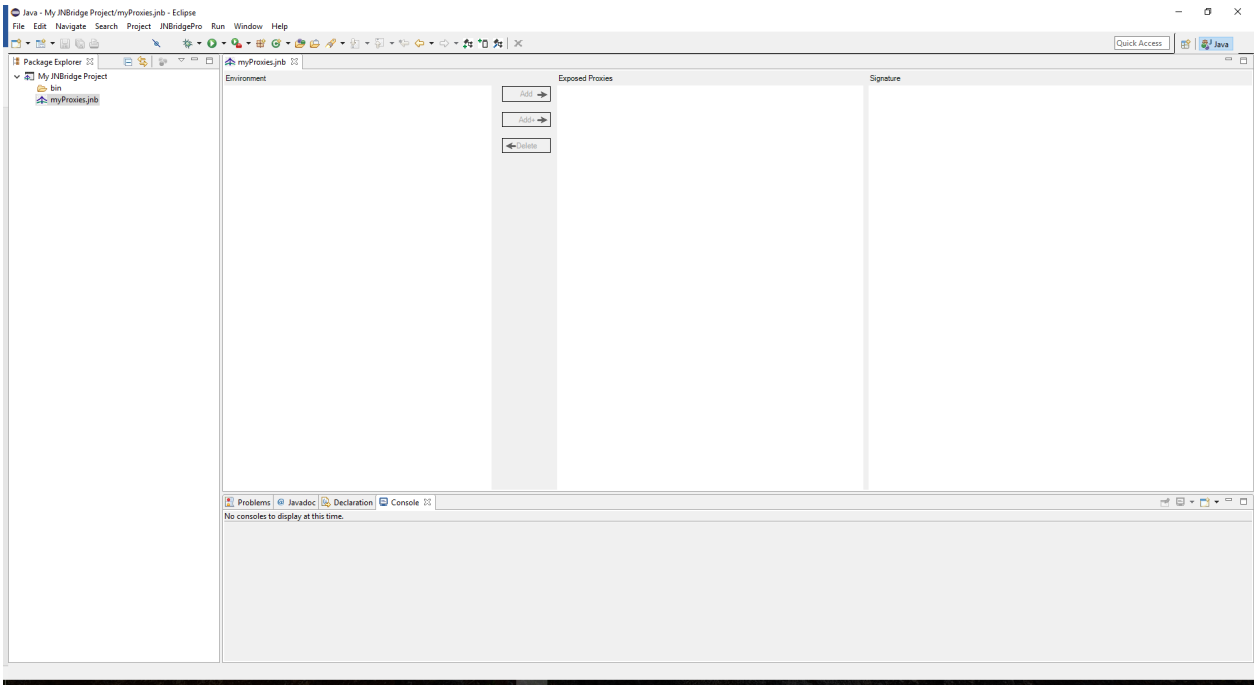


Figure 50. JNBProxy editor in Eclipse plug-in

Process for generating proxies

Generating proxies takes three steps:

- Add classes to the environment for which you might want to generate proxies.
- Select classes in the environment for which proxies are to be generated and add them to the exposed proxies list.
- Build the proxies into a .Java jar file.

Adding classes to the environment

The first step in generating proxies is to load candidate classes into the environment. Think of the environment as a palette from which one can choose the proxies that are actually generated. Classes can be loaded into the environment in two ways: from a .NET assembly DLL or EXE file, and as specific classes found among a set of assemblies.

To load classes from a .NET assembly, click on the menu item **JNBridgePro→Add classes from assembly file…**. A dialog box will appear allowing the user to locate one or more assembly files whose classes will be loaded. Opening an assembly file will cause all the classes inside of it to be added to the environment pane. The progress of the operation is shown in the output pane. The operation can be terminated before completion by clicking on the Stop button located on the GUI's tool bar.

Note: any assembly file from which classes are to be loaded must be in the assembly list. To edit the assembly list, see "Setting or modifying the assembly list," below.

Note: The results of operations terminated prematurely by clicking the stop button may be inconsistent and should not be used. These partial results can be removed by performing an Undo operation.

To load specific classes from the assembly list, click on the menu item **JNBridgePro→Add classes from assembly list…**. When this item is selected, the Add Classes dialog box is displayed (Figure 51).

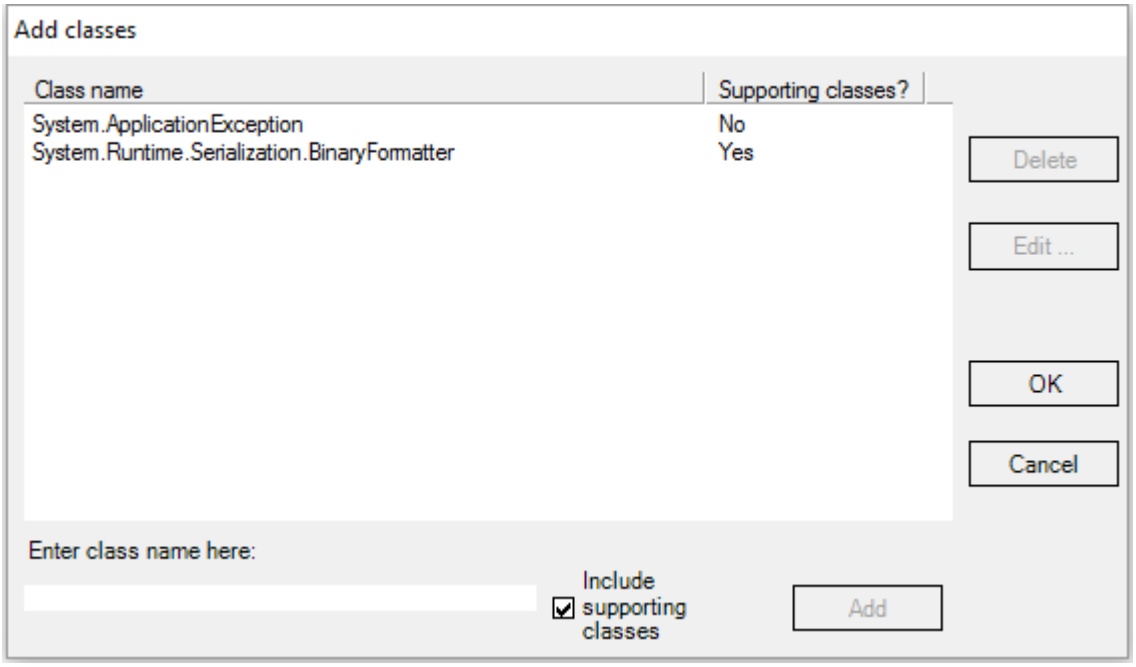


Figure 51. Add classes dialog box

To specify a class to be added, type the fully-qualified class name into the text box at the bottom of the dialog box. (Note that, as you type, the interface will provide class name suggestions based on what is available and what you have typed so far.) Leave checked the "Include supporting classes" check box to indicate that all classes supporting the specified class should be automatically added. (See *Supporting classes*, below, for more information on supporting classes.) Then, click the Add button to add the class to the list of classes to be loaded into the environment. You may delete a class from this list by selecting the class in the list and clicking on the Delete button. You may edit the information specified for the class by selecting the class and clicking on the Edit button or by double-clicking on the class. When you are done specifying classes, click the OK button to add the classes to the environment. This process may take a few minutes, especially if any of the classes in the list must also have their supporting classes loaded. The operation can be terminated before completion by clicking on the Stop button located on the GUI's tool bar.

To load a generic class into the environment, enter the class name using the format *className`numParmams*, where *className* is the name of the class without the generic parameters, *numParams* is the number of generic parameters, and where *className* and *numParams* are separated by a backquote (`). For example, to load System.Collections.Generic.List<T>, use the class name System.Collections.Generic.List`1. (Note that the name System.Collections.Generic.List<> will also be accepted, although we recommend using the backquote format.)

Also note that, in addition to specifying the fully-qualified name of a class to be added, you can also use a trailing '*' as a wildcard, to indicate that all classes in the assembly list that match begin with the string that precedes the asterisk should be added. For example, supplying 'System.Collections.*' indicates that all the classes in the System.Collections.* package should be added.

Note: If you are doing Java-.NET co-development, and a .NET class is changed after the class has been loaded into the environment (e.g., a method or field has been added or removed, or the signature changed), the class can be updated in the environment simply by loading it again.

Note: Any class or supporting class that is added must be in the assembly list or a member of the mscorlib.dll assembly.

Note: Proxies System.Object and System.Type are always generated and added to the environment even if they are not explicitly requested or implicitly requested by checking "Include supporting classes."

Note: The results of operations terminated prematurely by clicking the stop button may be inconsistent and should not be used. These partial results can be removed by performing an Undo operation.

Selecting proxies to be generated

Once classes have been loaded into the environment, they are displayed in the environment pane, which is a tree view listing all the classes loaded into the environment grouped by package. Next to each item in the tree view is an icon indicating whether the item is a package, an interface, and exception, or some other class. (See Figure 52).

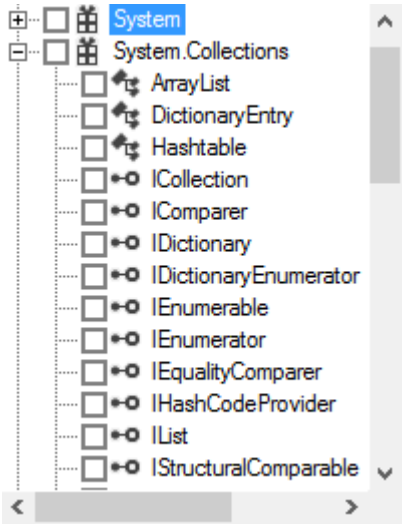


Figure 52. Environment pane

To select a class for proxy generation, check the class's entry in the environment pane by clicking on the check box next to the name. To select all the classes in a package, check the package's entry by clicking on its check box. Once a set of classes has been checked, add them to the set of exposed proxies by clicking on the **Add** button.

Alternatively, you may click on the **Add+** button to add the checked classes and all supporting classes. For a discussion on supporting classes, see the *Supporting Classes* section later in this guide. The checked classes and all supporting classes (if **Add+** was clicked) will appear in the exposed proxies pane, which is a tree view similar to the environment pane.

Note: use of* Add+ *may take a few minutes for the operation to complete. The operation can be terminated before completion by clicking on the Stop button located on the GUI's tool bar. The results of operations terminated prematurely by clicking the stop button may be inconsistent and should not be used. These partial results can be removed by performing an Undo operation.

To selectively remove classes from the exposed proxies pane so that proxies are not generated, check the entries in the exposed proxies pane for all classes or packages to be removed (by clicking on the check boxes next to the class or package names), then click on the **Delete** button. The checked items will be removed from the exposed proxies pane.

As a convenience, the JNBridgePro menu contains items to check or uncheck all the items in the environment or exposed proxy panes.

Add, **Add+**, and **Delete** operations may be repeatedly performed until the exact set of proxies to be generated appears in the exposed proxies pane.

The most recent **Add**, **Add+**, and **Delete** operations may be undone and then redone using the **Edit→Undo** and **Edit→Redo** menu items.

Note: Proxies for System.Object and System.Type will always be added to the exposed proxies pane when the Add button is clicked, even if they have not been checked in the environment pane. Also, System.Object and System.Type cannot be checked in the exposed proxies pane, and therefore cannot be deleted from that pane. The reason for this behavior is to ensure that proxies for Object and Type are always generated.

Designating proxies as reference or value

Before generating proxies for the classes listed in the exposed proxies pane, the user can designate which proxies should be reference and which should be value. (See the section "Reference and value proxies," below, for more information on the distinction between reference and value proxies.) The default proxy type is reference; if the user wants all proxies to be reference, nothing additional need be done.

To set a proxy's type (i.e., to reference or any of the three styles of value), position the cursor over the proxy class to be changed (in the exposed proxies pane), and right-click on the class. A pop-up menu will appear. Choose the desired proxy type. After the proxy type is selected, the proxy class will be color coded to indicate its type (black for reference, blue for value (public/protected fields style), red for by-value (JavaBean style), or green for by-value (mapped).

To set the types of multiple proxy classes at the same time, click on the **JNBridgePro→Pass by Reference / Value…** menu item. The Pass by Reference / Value dialog box is displayed (Figure 53). All proxy classes in the exposed proxies pane are listed in this dialog box. To change the types of one or more classes, select those classes (left-click on a class to select it, and shift-left-click or ctrl-left-click to do multi-selects), then click on the Reference, Value (Public/protected fields/properties), or Value (Mapped) button to associate the selected classes to the desired type. When done, click on the OK button to actually set the classes to the chosen types and dismiss the dialog box, click on the Apply button to set the classes without dismissing the dialog box, or click on the Cancel button to discard all changes.

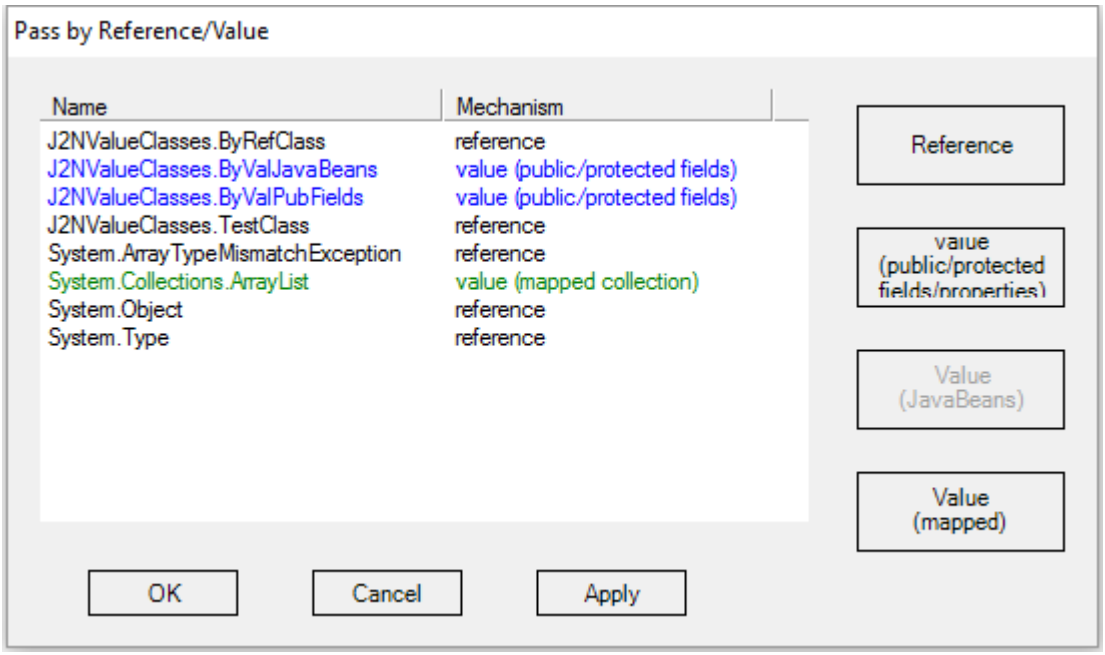


Figure 53. Pass by Reference/Value dialog box

Designating proxies as transaction-enabled (formerly "thread-true")

Before generating proxies for the classes listed in the exposed proxies pane, the user can designate which proxies should be *transaction-enabled*. (See the section "Transaction-enabled classes and support for transactions," below, for more information on the transaction-enabled classes.) The

default proxy type is not transaction-enabled; if the user wants all proxies to be non-transaction-enabled, nothing additional need be done.

To set a proxy as transaction-enabled, position the cursor over the proxy class to be changed (in the exposed proxies pane), and right-click on the class. A pop-up menu will appear. Select the "transaction enabled" menu item so that it is checked. To remove the transaction-enabled property from a proxy class, perform the same operation to uncheck the transaction-enabled menu item. It is currently not possible to make multiple classes transaction-enabled at the same time.

Users should use the transaction-enabled property sparingly, as it incurs a performance penalty. See the section "Transaction-enabled classes and support for transactions" for more information.

Generating the proxies

If Eclipse's Build Automatically option is set, a Java jar file containing proxies for the classes in the Exposed Proxies pane will automatically be generated whenever the contents of the Exposed Proxies pane changes. No additional action need be taken. If Build Automatically is not set, the proxy jar file can be generated by selecting the **Project→Build All** or **Project→Build Project** menu items. The proxy jar file will be placed in the bin folder inside the folder containing the associated JNBridge project.

Note: The Build operation may take a few minutes for the operation to complete. The operation can be terminated before completion by clicking on the Stop button located on the GUI's tool bar. The results of operations terminated prematurely by clicking the stop button may be inconsistent and should not be used.

Generating and using proxies as part of a larger build

It is possible to generate and use proxies as part of a larger build operation. To do so, simply edit the build path of the Java project that will use the proxies. In editing the Java Build Path, select the Libraries tab, then click on the "Add JARs…" button. A JAR Selection dialog box will be displayed. Navigate to the associated JNBridge project node, then into its bin directory and select the proxy jar file. One must also add the files jnbcore.jar and bcel-*6.n.m*.jar to the Java Build Path. Then, when one performs a build, the referencing project will use the proxy jar file generated by the JNBridge project. If the JNBridge project is out of date or has not yet been built, it will automatically be build or rebuilt before being used.

Information converning the generation of the proxies is displayed in Eclipse's Console window. If the build was unsuccessful, information describing the errors will be found there.

Saving projects

The current contents of the proxy generation editor is automatically saved to the .jnb file when the JNBridge project is built. The contents of the editor can be explicitly saved by selecting one of the **Save** items under the **File** menu. In addition, if the proxy generation editor is closed, and its contents has been modified since it was last saved, the user is offered the opportunity to save it.

The saved .jnb file can also be read and edited by the standalone JNBProxy GUI-based proxy generation tool.

Exporting a class list

Selecting the **JNBridgePro→Export classlist…** menu item will cause a text file to be generated that lists the classes in the Exposed Proxies pane, plus information on whether the classes are by-reference or

by-value, and whether the classes are transaction-enabled. The exported class list is in the correct format to be used as input with the '/f' option of the command-line version of JNBProxy. (See the section "Using JNBProxy (command-line version)").

Examining class signatures

JNBProxy displays information about a Java class so that a user can determine whether it contains properties of interest to the user. If a class item in either the environment or exposed proxies pane is selected, its information is displayed in the signatures pane. The information includes the class's name, superclass, attributes, interfaces, fields, properties, and method signatures (Figure 54).

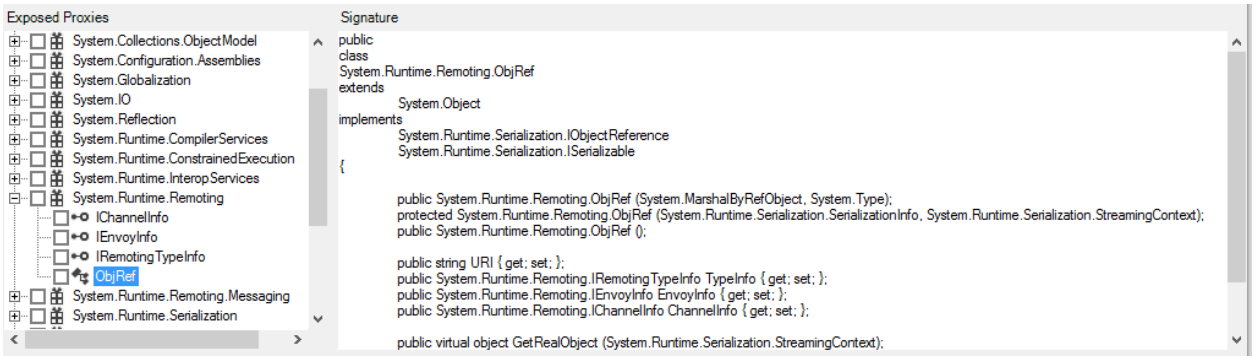


Figure 54. Selected item in exposed proxy pane and corresponding information in signature pane

Searching for classes in the environment or exposed proxy panes

To find a class among a large number of classes in the environment or exposed proxy panes, use the Find facility available by clicking on the **Edit→Find/Replace…** menu item, which displays a Find window (Figure 55). The Find window offers a variety of options, including the ability to search the environment or exposed proxy panes, to search down or up, to look for exact whole-word matches, and to perform case-dependent matches. To repeat a search, the user should continue using the **Edit→Find/Replace…** menu item.

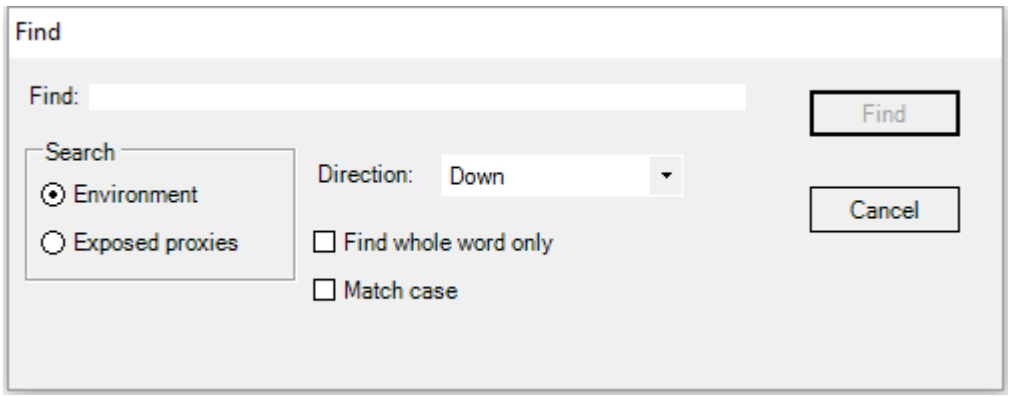


Figure 55. Find window.

Modifying tree view properties

To change the ways that the environment and exposed proxy panes are displayed, click on the **JNBridgePro→Tree View Options…** menu item. When this option is selected, a dialog box is displayed that allows the user, for either the environment or exposed proxy pane, to show or hide the icons in the pane, and to completely expand or completely collapse the tree view in the pane (Figure 56).

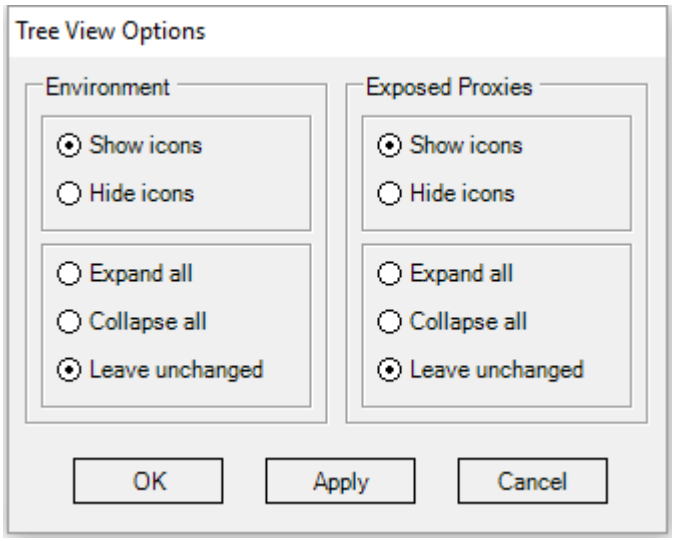


Figure 56. Tree View Options window

Refreshing the display

To refresh the contents of the three panes (environment, exposed proxy, and signature) in the JNBProxy window, select the **JNBridgePro→Refresh** menu item.

Generating Java SE 5.0 (and later)-targeted proxies

By default, generated Java-side proxies are compatible with Java SE 5.0 and later, and can therefore take advantage of mappings from .NET features such as generics, enums and vararg methods, to the equivalent Java features, all of which were introduced with Java SE 5.0. If you wish to target the generated proxies to JDK 1.1 (and therefore make them compatible with 1.1 through 1.4), you must explicitly turn off this option.

To make sure that proxies are targeted toward Java SE 5.0, bring up the Java Options window (Figure 57) by selecting Windows→ Options…, navigating to the JNBridgePro section, and selecting the Java Options tab, and check the "Generate Java SE 5.0-targeted proxies" check box. This box is checked by default. To target the proxies to earlier versions of Java, uncheck the box.

Note: Java-side proxies targeted to Java SE 5.0 will not work with earlier versions of Java. However, you can use earlier versions of Java to generate Java SE 5.0-targeted proxies.

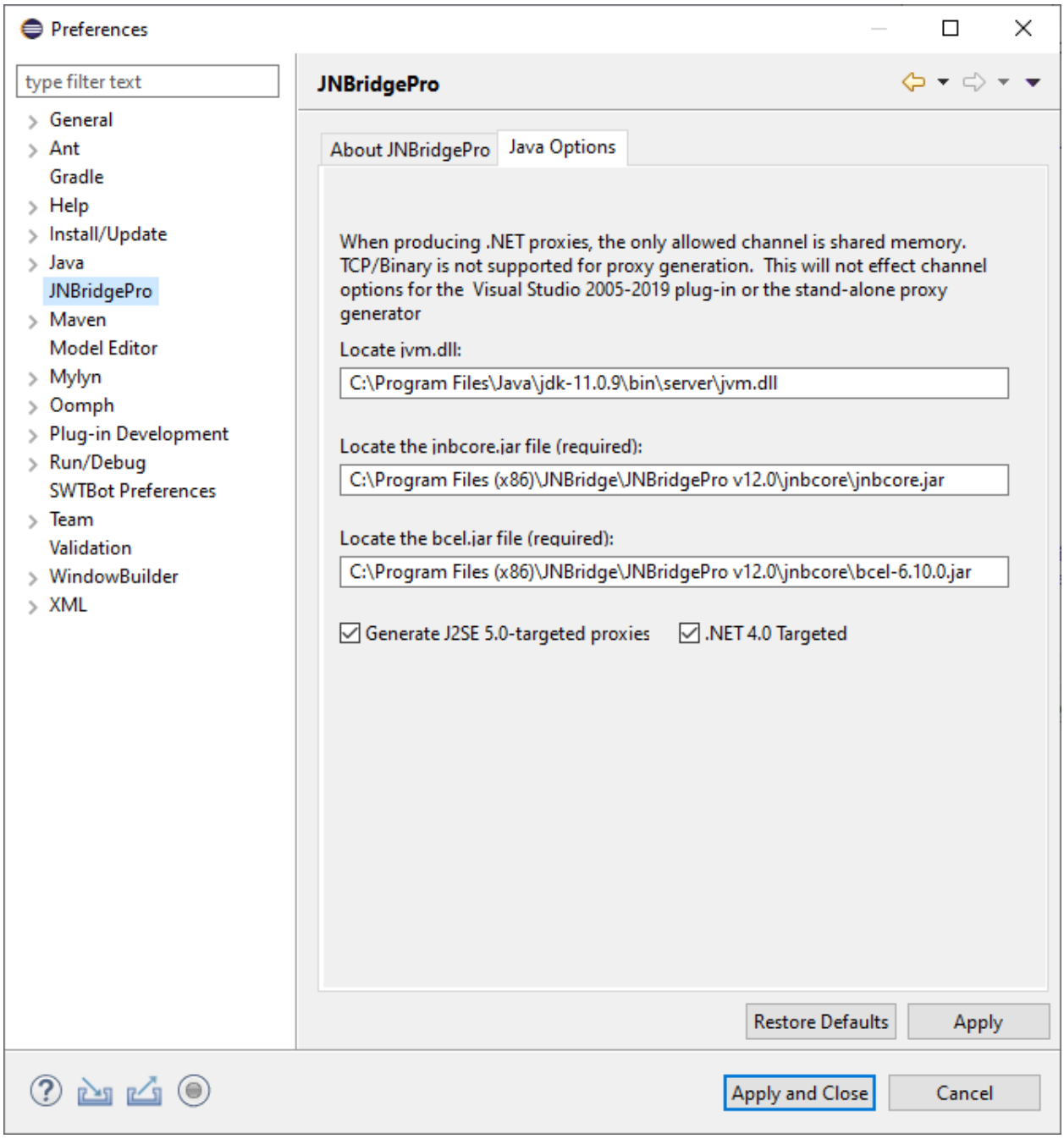


Figure 57. Java Options window

Setting or modifying the assembly search path

Before loading .NET types for a Java-to-.NET project, the user needs to specify the assemblies from which these types will be loaded. The **JNBridgePro→Edit Assembly List…** menu item brings up an **Edit Assembly List** dialog box (Figure 58), similar to the **Edit Classpath** dialog box in .NET-to-Java projects. Added assemblies can be either EXE (executable) or DLL files, but they must be .NET assemblies. If left empty, classes can only be loaded from the standard mscorlib.dll assembly.

Assemblies can be added to the assembly list from specific file locations, or they can be added to the assembly list from the Global Assembly Cache (GAC). To add an assembly from the GAC, click the **Add from GAC…** button in the **Edit Assembly List** dialog box. A **Select Assemblies from GAC…** dialog box will be displayed (Figure 59). Simply select the assemblies to be added (one can select multiple assemblies by shift-clicking or ctrl-clicking on the assemblies in the displayed list), then click on the OK button to add the selected assemblies to the assembly list. Note that different versions of the same assembly can be in the GAC and will be displayed in the list. The user should be careful to select the desired version when adding assemblies from the GAC.

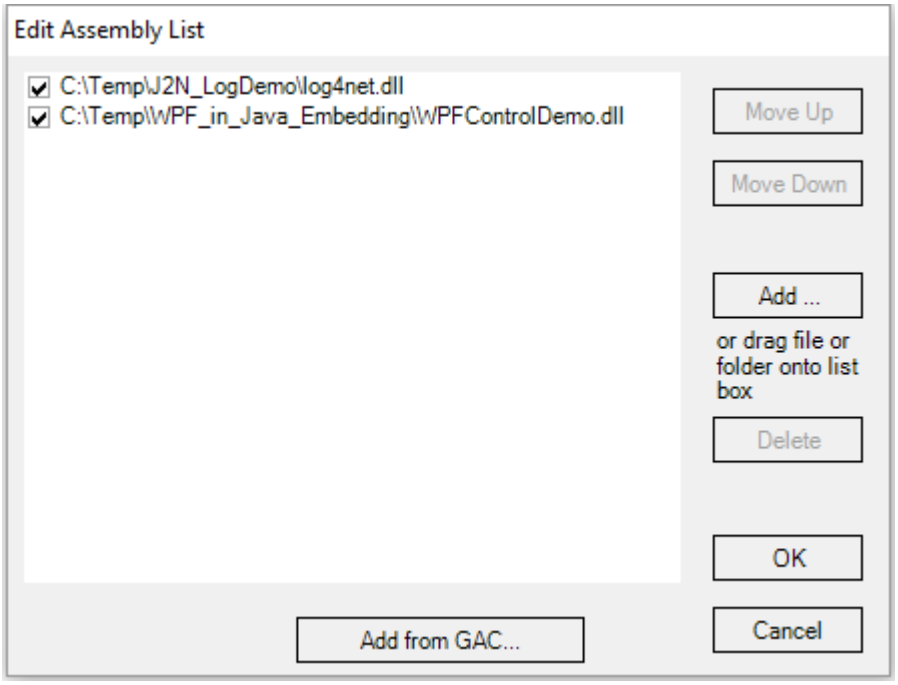


Figure 58. Edit Assembly List dialog box.

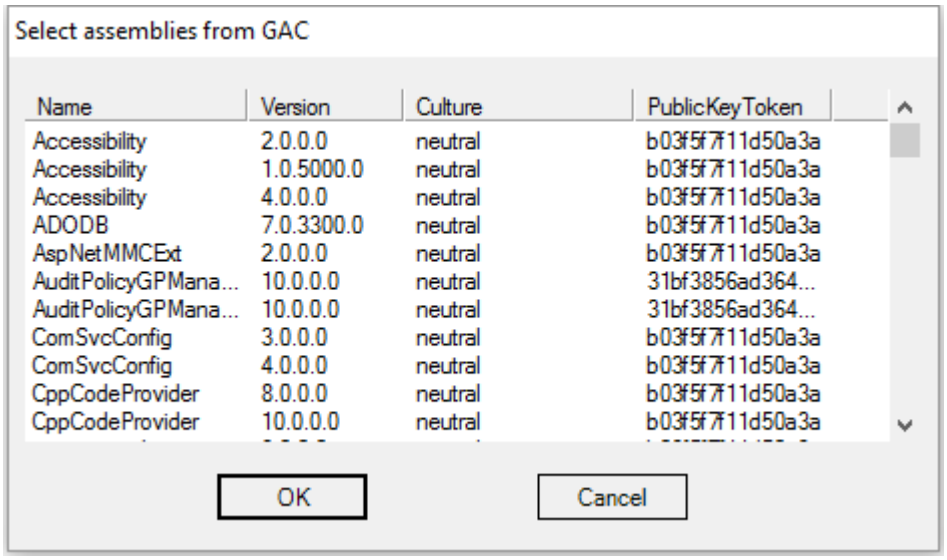


Figure 59. Select Assemblies from GAC dialog box

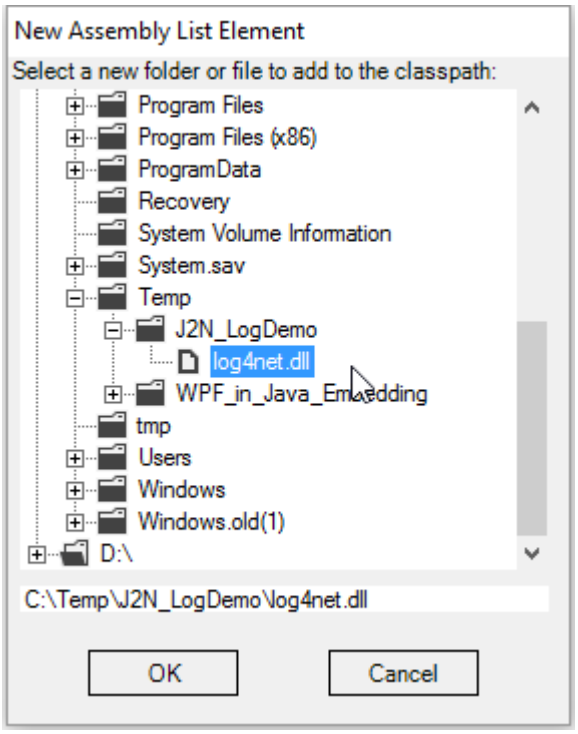


Figure 60. New Assembly List Element window.

The assembly list can be rearranged by selecting an assembly element and clicking on the **Move Up** or **Move Down** buttons, or by selecting an element and clicking on the **Delete** button. Note that only checked elements will be added to the assembly list when the dialog is dismissed.

If a type appears in more than one assembly in the assembly list, then its definition will be taken from the first assembly in the list in which the type is defined.

To add a folder or jar file to the classpath, simply drag and drop one or more files or folders onto the Edit Classpath form. You can also add a folder or jar file by clicking on the **Add…** button. This causes a New Assembly List Element window to be displayed. In this window, the user can navigate to the desired folders or assembly files, or can enter a file path directly (Figure 60). The New Assembly List Element window supports multiple selection - multiple folders and/or assembly files may be selected by ctrl-clicking, while a range of folders and/or assembly files may be selected by shift-clicking. Clicking on the **OK** button will cause the indicated folders or files to be added to the Edit Assembly List window. Selecting a folder will cause all the assemblies in the folder to be added to the assembly list.

Note: You can also type a directory or file path directly into the New Assembly List Element dialog box. This path can be an absolute path, or a UNC path to a shared network folder (e.g., \\MachineName\FolderName\FileName.dll).

Choosing the .NET platform (platform targeting)

In earlier versions of JNBridgePro you could choose whether to load the classes into the .NET 2.0 runtime (used by .NET 2.0/3.0/3.5-targeted assemblies) or into the .NET 4.0/4.5/4.6/4.7/4.8 runtime. This choice is no longer applicable and only .NET 4.8 can be targeted. See the Window→Preferences… menu item, then select the JNBridgePro preferences and choose the "Java Options" tab. You will see a checkbox labeled ".NET 4.0 Targeted" (Figure 61). This actually refers to .NET 4.8 and this checkbox should remain checked. The assemblies will be loaded in the .NET 4.8 runtime.

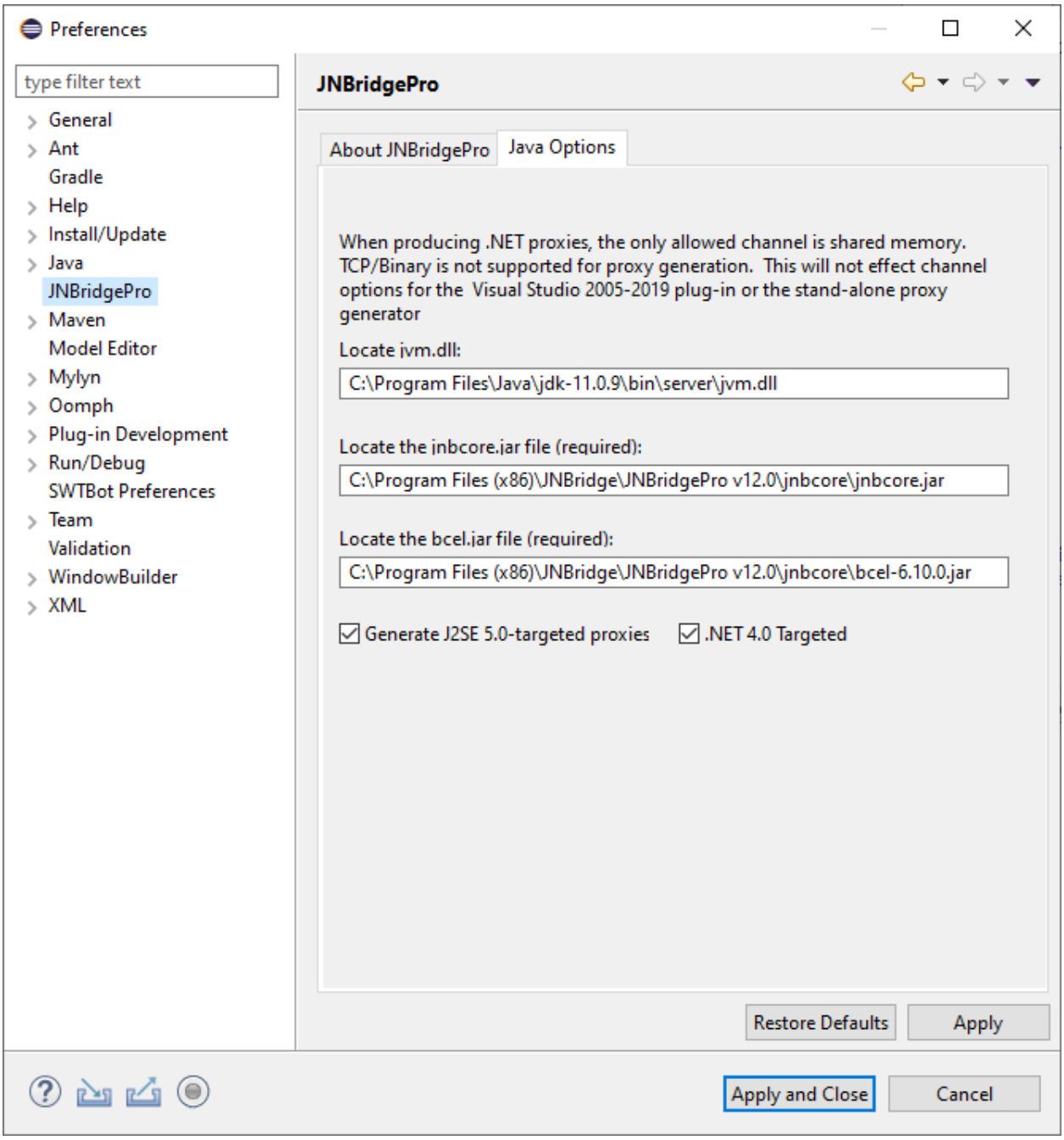


Figure 61. Choosing a .NET platform

Using JNBProxy (command-line version)

Use the **JNBProxy** command-line application (jnbproxy.exe) to generate Java proxies for .NET classes. Unless the /nj option is used (see below) and the Java side is started manually, a copy of java.exe (or the folder in which it resides) must be found in the system execution path. If it is not, and java.exe is not found, an error will be reported and JNBProxy will terminate. (Note, only use jnbproxy32.exe when you are running on 64-bit systems, and you are proxying classes from 32-bit DLLs. For all other situations, use jnbproxy.exe.)

JNBProxy is used from the command line as follows:

jnbproxy <options> *classes*...

classes… — A space-separated sequences of fully qualified .NET class names (e.g., `System.String`) for which proxies should be generated. Note that proxies for `System.Object` and `System.Type` are always generated, even if they are not listed in the class list. To generate a proxy for a generic class, use the class name format *className`numParmams*, where *className* is the name of the class without the generic parameters, *numParams* is the number of generic parameters, and where *className* and *numParams* are separated by a backquote (`). For example, to load System.Collections.Generic.List<T>, use the class name System.Collections.Generic.List`1. (Note that the name System.Collections.Generic.List<> will also be accepted, although we recommend using the backquote format.)

Options

/al *assemblylist* — Assembly list. Used in Java-to-.NET projects, *assemblylist* is a semicolon-separated series of file paths of .NET assemblies (DLL and EXE files) containing the .NET types with which the Java classes will communicate.

/bp *bcelpath* — BCEL classpath. *bcelpath* is the file path locating the folder containing the file `bcel-6.n.m.jar`. If left out, the environment variable CLASSPATH is used to locate `bcel-6.n.m.jar`. If using shared-memory communications, this option must be supplied.

/d *directory* — Directory. Write the generated proxies to jar file in the specified *directory*. If left out, the default is the current execution directory.

/f *classfile* — Read from file. Read classes for which proxies are to be generated from the specified text file rather than from the command line. The purpose of this option is to allow users to generate large numbers of proxies where it would be impractical to type them onto the command line. The file must consist of a list of fully-qualified class names, one per line. Optionally, the class name can be followed by white space (one or more spaces and/or tabs), followed by the letter "r" (denoting that the class should be a reference proxy – see "Reference and Value Proxies," below), "f" (denoting that the class should be a public/protected fields/properties-style value proxy), or "m" (denoting that the class should be a directly-mapped value proxy). In addition to the above, the class name can also be followed by the letter "t", indicating that the class should be *transaction-enabled* (see "Transaction-enabled classes and support for transactions," below). Any other trailing letter, or no letter, results in the generation of a reference

	proxy. If the /f option is used, any classes listed on the command line will be ignored.
/h	Help. List the options and usage information. All other options and arguments will be ignored. Simply typing the command *jnbproxy* with no options or arguments will result in the same information
/host *hostname*	Host. The host on which the Java side will be found. Can be a name or an IP address. Can be "localhost". Required if the /pro option is used, and its value is "b" or "h". Ignored otherwise.
/java *javapath*	Java path. *Javapath* is the file path locating the java.exe to be used when autostarting Java. If left out, the first java.exe in the system PATH environment variable will be used.
/jp *jnbcorepath*	JNBCore classpath. *jnbcorepath* is the file path locating the folder containing the file `jnbcore.jar`. If left out, the environment variable CLASSPATH is used to locate `jnbcore.jar`. If using shared-memory communications, this option must be supplied.
/jvm *jvmpath*	JVM. The full path of the jvm.dll file used in shared-memory communications. See *Shared-memory communications* for more information. Required for shared-memory communication, ignored with all other communications mechanisms.
/ll *licenseFilePath*	License location. Provides the path to the ***folder*** containing the license file. If this option is omitted, the license file must be in the same folder as jnbproxy.dll. ***(Only recognized when used with the .NET Core-targeted proxy tool.)***
/ls	List all classes to be generated in support of classes (see *Supporting classes*, below), but don't actually generate the proxies. If left out, the proxies will be generated as well as listed.
/n *name*	Name. *name* is the name of the jar file containing the proxies. If left out, a file `jnbproxies.jar` will be created to contain the proxies.
/nj	No Java. If option is present, the Java-side must be started manually, and the /cp, /jp, and /xp options, if present, are ignored. If left out, the Java-side is started automatically.
/ns	No supporting classes. If option is present, jnbproxy will only generate proxies for the classes specified on the command line, but not for any supporting classes. If left out, proxies for all supporting classes will be generated.
/nt50	Do not target Java SE 5.0. If option is present, then Java-side proxies targeted to JDK 1.3 will be generated. If the option is missing, the proxies will be targeted towards Java SE 5.0, and will therefore generate Java-side proxies that support mappings from .NET generics, enums, and vararg methods. The option is ignored in .NET-to-Java projects.
/pd *proxyDirection*	Proxy direction. Specifies the direction in which the proxies will operate. *proxyDirection* may be either **n2j**, in which case it is a .NET-to-Java project and .NET-side proxies in a DLL file will be generated to call Java classes, or **j2n**, in which case it is a Java-to-.NET project and Java-side proxies in a jar file will be generated to call .NET types. If some other proxy direction is supplied, or the /pd option is omitted, the direction will be assumed to be **n2j**.

/port *portNum*	Port. The port on which the Java side will be listening. Must be an integer. Required if the /pro option is used, and its value is "b" or "h". Ignored otherwise.
/pro *protocol*	Protocol. Specifies the protocol/communication mechanism to be used to communicate between the .NET and Java sides. *protocol* can be either **s** (shared memory), or **b** (binary). If **s**, the /jvm, /bp, and /jp options are required; /nj, /host, and /port are ignored. If **b**, the /host and /port options are required; /jvm is ignored. If the /pro option is omitted, /jvm, /host, and /port are all ignored and the .NET-side configuration file jnbproxy.config is used to set up the communications. (Use of jnbproxy.config is supplied for backward compatibility with earlier versions.)
/pp *propspath*	Property path. *propspath* is the file path for the folder containing the file `jnbcore.properties`. If left out, the value *jnbcorepath* in the /jp option will be used.
/wd *dir*	Working directory. *dir* is the working directory for the Java-side's JVM. This option is ignored if the /nj option is present (that is, if the Java-side is being started manually. If left out, the system's default working directory will be used.

Any other options, specifically those pertaining to .NET-to-Java projects, will be ignored when the /pd option is **j2n** and a Java-to-.NET project is being generated.

Supporting classes

JNBProxy can generate proxies not only for the .NET classes that are explicitly listed, but also for *supporting classes*. Informally, a supporting class for a given .NET class is any class that might be needed as a direct or indirect result of using that .NET class. Formally, for a given .NET class, supporting classes include all of the following:

- The given class itself.
- The class's superclass or superinterface (if it exists) and all of its supporting classes.
- The class's implemented interfaces (if any) and all of their supporting classes.
- For each field in the class, the field's class and all of its supporting classes.
- For each property in the class:
  - the property's class and all of its supporting classes.
  - for each of the property's index parameters, the parameter's class and all of its supporting classes.
- For each method in the class:
  - The method's return value's class (if any) and all of its supporting classes.
  - For each of the method's parameters, the parameter's class and all of its supporting classes.
- For each constructor in the class:
  - For each of the constructor's parameters, the parameter's class and all of its supporting classes.

Note that, unlike Java, where supporting classes include exceptions declared to be thrown by methods, .NET supporting classes don't include thrown exceptions, since they are not declared in advance.

The number of supporting classes depends on the classes explicitly listed, but will probably be on the order of 200-250 classes. It is recommended that all supporting classes be generated, although there are situations where you, to save time or space, may choose to generate only those classes explicitly specified, without supporting classes.

If a proxy for a supporting class has not been generated, and a proxy for such a class is later needed when the proxies are used, the proxy for the nearest superclass to the required class is used instead. For example, if a proxy for a class C is needed, and the proxy has not been generated, the proxy for the superclass of C will be used if it has been generated. If that proxy hasn't been generated, the proxy for that superclass's superclass will be used if it has been generated, and so forth, until the proxy for `System.Object` (which is always generated) is encountered. Thus, even with an incomplete set of proxies, code will remain functional, although functionality and other information may be lost.

In the GUI version of JNBProxy, **Add** will expose just the explicitly listed classes, whereas **Add +** will additionally expose the supporting classes. In the command line version of JNBProxy, the default is to generate proxies for the supporting classes: use of the /ns option will override this default.

Starting Java manually

JNBProxy uses the Java reflection API to discover information about the 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 which proxies are to be generated.

Both the GUI and command-line versions of JNBProxy can be invoked so that they automatically start and stop the Java-side. In some situations, however, it may be necessary to manually start up the Java-side. For example, the Java-side may be on some other machine, in which case JNBProxy is unable to start it up.

To manually start up the Java-side, the following command-line command must be given:

java –cp *classpath* com.jnbridge.jnbcore.JNBMain /props *propFilePath*

where *classpath* must include:

- The Java classes for which proxies are to be generated (and their supporting classes)
- `jnbcore.jar`
- `bcel-`*`6.n.m`*`.jar`

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`.

Alternatively, one can specify one or more properties on the command-line using the /p option:

java –cp *classpath* com.jnbridge.jnbcore.JNBMain /p *name1=value1* /p *name2=value2* …

where each "/p" precedes a name/value properties pair (for example, */p javaSide.serverType=tcp*). The /p option allows individual Java-side properties to be supplied on the command line.

One can also specify both a properties file and individual command-line properties. In such a case, the properties file is read first, then the command-line properties are read, and override any properties of the same name in the properties file.

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

When the Java-side is started manually, an output similar to the following will be seen if the binary protocol is being used:

```
JNBCore v12.0
Copyright 2019, JNBridge, LLC

creating binary server
```

If a process (for example, another Java side) is already listening on the new Java side's port, the new Java side will print out an error message and terminate.

Proxy generation example

Assuming that we have the following C# classes, both in the com.jnbridge.samples package, and located in the assembly samples.dll:

```
public class Person
{
        public string name;
        public int getID(){…};
        public Person(string name){…};
}

public class Customer : Person
{
        public int getLastOrder(){…};
        static public int getLastCustomerID(){…};
        public Person getContact(){…};
        public void setContact(Person p){…};
        public void addOrder(string orderInfo){…};
        public bool customerStatus();
        static public void registerCustomer(Customer c){…};
        public Customer(){…};
}
```

Assuming that CLASSPATH is properly set (so that it includes `jnbcore.jar, bcel-6.n.m.jar`), issuing the command

```
jnbproxy /pd j2n /al samples.dll /pp propspath com.jnbridge.samples.Customer
```

(where *propspath* is the file path for the folder containing the file `jnbcore.properties,` and `samples.dll` is in the current directory)

will result in jar file `jnbproxies.jar` being created in the current directory containing proxies for com.jnbridge.samples.Customer, com.jnbridge.samples.Person, and their supporting classes. Note that any exceptions thrown by the methods are not considered supporting classes and must be explicitly specified.

Proxy generation with .NET Core

To generate Java proxy JAR files for Java-to-.NET Core projects, we provide a proxy generation tool designed to work with .NET Core. This proxy generation tool is command-line only. (.NET Core 3.0 does support Windows Forms and WPF applications, but only on Windows, and we wanted this tool to work on all platforms supported by .NET Core and Java.) This command-line-based proxy generation tool works in the same way as the command-line-based "traditional" proxy generation tool in the Java-to-.NET direction. If you attempt to use this proxy generation tool to generate proxy DLLs for .NET Core-to-Java projects, it will display an error message.

The proxy tool uses the same command-line options as the .NET Framework-targeted proxy tool (except that only /**pd j2n** is recognized). There is one additional option:

/ll *licenseFilePath* License location. Provides the path to the ***folder*** containing the license file. If this option is omitted, the license file must be in the same folder as jnbproxy.dll.

Using proxies with JNBridgePro

System configuration for proxy use

Prior to using the proxies, the system must be properly configured for their use. This section describes the necessary components and the various configuration details.

.NET-side

Since the .NET Framework supports distributed computing, it is possible that the .NET-side may reside on several machines, where the classes communicate with each other through .NET remoting. Every machine on the .NET-side that communicates with the Java-side must have a local copy of `jnbshare.dll`, and the generated proxy DLL file. Note that every copy of `jnbshare.dll` must be licensed. In addition, each .NET-side must have some method for configuring .NET-side communications. See "Configuring the .NET-side," below.

If shared-memory communication is being used, jnbsharedmem (jnbsharedmem_x86.dll, jnbsharedmem_x64.dll, or both), jnbjavaentry (jnbjavaentry_x86.dll, jnbjavaentry_x64.dll, or both), and jnbjavaentry2 (jnbjavaentry2_x86.dll, jnbjavaentry2_x64.dll, or both) must also reside on the machine. For Java-to-.NET calls using shared memory, jnbshare.dll, jnbsharedmem (jnbsharedmem_x86.dll, jnbsharedmem_x64.dll, or both), and jnbjavaentry2 (jnbjavaentry2_x86.dll, jnbjavaentry2_x64.dll, or both) must all be installed in the Global Assembly Cache (GAC), or in the folder specified by the dotNetSide.appBase property (see "Specifying the .NET-side application base folder"). This includes situations in which bidirectional shared-memory communication is used. See the .NET documentation on gacutil.exe for information on how to install assemblies in the GAC. ***If you are not performing Java-to-.NET calls using shared memory, it is not necessary to install these assemblies in the GAC or in the dotNetSide.appBase folder.***

Java-side

There must be a copy of the Java Runtime Environment (JRE), residing on the Java-side. If the Java-side resides on more than one machine, each machine must have a copy of the JRE. The Java-side must also include the Java classes being accessed, and must have a copy of the file `jnbcore.jar` on

each machine on which the Java-side resides. The locations of the Java classes and `jnbcore.jar` must either be in the CLASSPATH environment variable, or must be supplied to the Java runtime when it is invoked.

The Java-side properties must somehow be specified. This can be done through the properties file `jnbcore.properties`, through the specification of individual properties on the command-line, or programmatically, through a Properties object supplied as a parameter in a call to JNBMain.start(). See below for more information.

Configuring the .NET side

The .NET side is configured through the application's configuration file, or programmatically in the user's .NET code. In older versions of JNBridgePro, the .NET-side was configured through use of the special configuration file jnbproxy.config. While use of jnbproxy.config has been deprecated, it is still supported for backward compatibility. Details of jnbproxy.config and its use are given an appendix at the end of the document.

Note: For Java-to-.NET projects using shared memory, the .NET side does not need to be explicitly configured using the methods below. All configuration of the JNBridgePro .NET side will be handled through the Java-side configuration.

Configuring communications through the application configuration file. The application configuration file is the file that is named *app*.exe.config (for an application *app*.exe) or web.config (if it is a configuration file for an ASP.NET Web application).

Inside the <configuration> section of the application configuration file, add the following section if it is not already there:

```
<configSections>
   <sectionGroup name="jnbridge">
   <section name="dotNetToJavaConfig"
              type="System.Configuration.SingleTagSectionHandler" />
   <section name="javaToDotNetConfig"
              type="System.Configuration.SingleTagSectionHandler" />
   <section name="tcpNoDelay"
              type="System.Configuration.SingleTagSectionHandler" />
   <section name="javaSideDeclarations"
              type="System.Configuration.NameValueSectionHandler" />
   <section name="assemblyList"
              type="com.jnbridge.jnbcore.AssemblyListHandler, JNBShare,
Version=12.10.0.0, Culture=neutral, PublicKeyToken=b18a44fb28aea122" />
   </sectionGroup>
</configSections>
```

Note that for historical reasons, for the 4.8-targeted version of JNBridgePro being used, the "assemblyList" definition above should refer to "JNBShare, Version=12.10.0.0," not "12.0.0.0".

If there is already a <configSections> section, simply add the <sectionGroup> and <section> tags above to the <configSections> section.

Inside the <configuration> section of the application configuration file, add the following section:

```
<jnbridge>
  … .NET-side specification goes here …
</jnbridge>
```

To configure a .NET-side server's communications (for Java-to-.NET communication), add the following elements inside the <jnbridge> section:

```
<javaToDotNetConfig scheme="jtcp"
                    port="local port number"
                    useSSL="true or false"
                    certificateLocation="file path of server certificate"
                    useIPv6="true or false"/>
```

The *useIPv6* attribute in the <javaToDotNetConfig> element indicates whether the .NET side should implement an IPv6 listener. If the value is true the .NET side will listen on any IPv6 address on the local machine, otherwise, the .NET side will listen on any IPv4 address. The attribute is optional; if it is omitted, it is equivalent to *useIPv6="false"*.

Note: Use of IPv6 is only supported when running on operating systems that support IPv6. If the underlying operating system does not support IPv6 and *useIPv6* is set to true, an exception will be thrown.

The *useSSL* attribute in the <dotNetToJavaConfig> and <javaToDotNetConfig> elements indicates whether secure communications using SSL should be used. If the value is true, SSL will be used, otherwise it will not be used. The attribute is optional; if it is omitted, it is equivalent to *useSSL="false"*.

The *certificateLocation* attribute in the <javaToDotNetConfig> element indicates the location of the server SSL certificate. It is only required when *useSSL* is *true*. If *useSSL* is *false*, the *certificateLocation* attribute is ignored and may be omitted.

For more information on secure communications, see the sections on "Secure communications using SSL" in the .NET-to-Java and Java-to-.NET sections of the *Users' Guide*.

To tell a .NET-side server that Java clients will access class in certain assemblies, add the following elements inside the <jnbridge> section:

```
<assemblyList>
   <assembly file="path to assembly or fully qualified name"/>
   ...
</assemblyList>
```

A bare-bones application configuration file declaring a.NET-side tcp server configuration (communicating via binary communication and listening on port 8086), and whose Java clients will access classes from the assembly C:\F\x.dll and System.Windows.Forms from the GAC, would look as follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <sectionGroup name="jnbridge">
   <section name="dotNetToJavaConfig"
            type="System.Configuration.SingleTagSectionHandler" />
   <section name="javaToDotNetConfig"
            type="System.Configuration.SingleTagSectionHandler" />
   <section name="tcpNoDelay"
            type="System.Configuration.SingleTagSectionHandler" />
   <section name="javaSideDeclarations"
            type="System.Configuration.NameValueSectionHandler" />
   <section name="assemblyList"
            type="com.jnbridge.jnbcore.AssemblyListHandler, JNBShare,
Version=12.10.0.0, Culture=neutral, PublicKeyToken=b18a44fb28aea122" />
   </sectionGroup>
   <jnbridge>
      <javaToDotNetConfig scheme="jtcp" port="8086" />
```

```
      <assemblyList>
         <assembly file="C:\F\x.dll" />
         <assembly file="System.Windows.Forms, Version=1.0.5000.0, Culture=neutral,
           PublicKeyToken=b77a5c561934e089" />
      </assemblyList>
 </jnbridge>
</configuration>
```

Note that for historical reasons, for the 4.8-targeted version of JNBridgePro being used, the "assemblyList" definition above should refer to "JNBShare, Version=12.10.0.0," not "12.0.0.0".

Note that either or both .NET-side client and server information may appear in the application configuration file, depending on whether the user is configuring .NET-to-Java calls, Java-to-.NET calls, or both.

If communications configuration appears in the application configuration file, it will override any communication configuration information in jnbproxy.config.

Configuring communications programmatically. .NET-side communications can also be configured programmatically in the user's code. To do so, call one of the overloads of the method

static void com.jnbridge.jnbproxy.JNBRemotingConfiguration.specifyRemotingConfiguration();

There are various ways to call specifyRemotingConfiguration() in Java-to-.NET and bidirectional contexts:

- specifyRemotingConfiguration(JavaScheme *localProtocol*, int *localPort*, bool *useSSL*, string *serverCertificateLocation*, string *serverCertificatePassword*, bool *useIPv6*, string *hostIP*, string[] *ipWhitelist*, bool *useClassWhiteList*, string *classWhiteListFile*)

  is used to specify a .NET-side server configuration (for Java-to-.NET communication), where *localProtocol* is the protocol used to communicate with the Java side, and *localPort* is the port on which the .NET side will listen for requests from the Java side. *useSSL* should be true if SSL (Secure Sockets Layer) should be used for the communications, and should be false otherwise. *serverCertificateLocation* is the file path of the X.509 certificate to be used by the server if SSL is to be used. If *useSSL* is false, *serverCertificateLocation* and *serverCertificatePassword* are ignored and can be null. (See the sections "Secure communications using SSL" in the .NET-to-Java and Java-to-.NET sections of the *Users' Guide* for more information; information on *serverCertificateLocation* is in the Java-to-.NET section, since it concerns Java-to-.NET communications.) *useIPv6* should be true if the .NET side should listen to requests on any IPv6 address on the local machine; if it is false, the .NET side should listen for requests to any IPv4 address on the local machine. Note that JavaScheme is the enumerated type com.jnbridge.jnbproxy.JavaScheme, and must have the value JavaScheme.binary.

  hostIP represents the IP address to be bound to the .NET side. It will only respond to requests addressed to that IP address. If it is set to "*", the .NET side will respond to requests addressed to any IP address associated with that machine. See "Binding IP addresses," for more information.

  ipWhitelist represents a list of IP addresses or ranges from which the .NET side will accept requests. When set to string[]{"*"}, it will accept requests from any Java client. *Note that ipWhitelist only works with TCP/binary communications.*

  useClassWhiteList is used to indicate whether class whitelisting is to be used. See "Class whitelisting" for more information. If it is true, *classWhiteListFile* indicates the path to the class

whitelist file. If *useClassWhiteList* is false, the value supplied for *classWhiteListFile* is ignored, and can be null

Note: Use of IPv6 is only supported when running on operating systems that support IPv6. If the underlying operating system does not support IPv6 and *useIPv6* is set to true, an exception will be thrown.

- specifyRemotingConfiguration(JavaScheme *remoteProtocol*, String *remoteHost*, int *remotePort*, JavaScheme *localProtocol*, int *localPort*, bool *useSSL*, string *serverCertificateLocation,* bool *useIPv6*, string[] *alternateServerNames*, string *clientCertificateLocation*, string *clientCertificatePassword*, string *clientCertificatePasswordFileLocation*, string *localHostIP*, string[] *localIPWhitelist*, bool *useLocalClassWhiteList*, string *localClassWhitelistFile*)

  is used to specify both a .NET-side client configuration (for .NET-to-Java communication) and a .NET-side server configuration (for Java-to-.NET communication), where *remoteProtocol* is the protocol with which the .NET side will be sending calls to the Java side, *remoteHost* is the name or IP address of the machine on which the Java side resides (if it is the same machine, it can be "localhost"), *remotePort* is the port on which the Java side is listening for requests, *localProtocol* is the protocol with which the Java side will be sending calls to the .NET side, and *localPort* is the port on which the .NET side will listen for requests from the Java side. *useSSL* should be true if SSL (Secure Sockets Layer) should be used for the communications, and should be false otherwise. *serverCertificateLocation* is the file path of the X.509 certificate to be used by the server if SSL is to be used. If *useSSL* is false, subsequent SSL-oriented parameters (*serverCertificateLocation*, *alternateServerNames*, *clientCertificateLocation*, *clientCertificatePassword*, *clientCertificatePasswordFileLocation*) are ignored. (See the sections "Secure communications using SSL" in the .NET-to-Java and Java-to-.NET sections of the *Users' Guide* for more information on these parameters and their use.) *useIPv6* should be true if the .NET side should listen to requests on any IPv6 address on the local machine; if it is false, the .NET side should listen for requests to any IPv4 address on the local machine. *localHostIP* specifies the IP address to which the .NET side should be bound when it responds to requests from a Java side. (See "Binding an IP address.") *localIPWhitelist* is the list of IP addresses of Java clients that are allowed to make calls to the .NET side. (See "Whitelisting Java clients.") *useLocalClassWhiteList* specifies whether class whitelisting should be used on the .NET side in Java-to-.NET calls, and (if *useLocalClassWhiteList* is true) *localClassWhiteListFile* is the path to the file containing the class whitelist. (See "Class whitelisting.") Note that JavaScheme is the enumerated type com.jnbridge.jnbproxy.JavaScheme, and must have the value JavaScheme.binary.

  Note: Use of IPv6 is only supported when running on operating systems that support IPv6. If the underlying operating system does not support IPv6 and *useIPv6* is set to true, an exception will be thrown.

- specifyRemotingConfiguration(JavaScheme *remoteProtocol*, String *jvmLocation*, String *jnbcoreLocation*, String *bcelLocation*, String *classpath*, String[] *jvmOptions*, String *javaEntryLocation*)

  is used to configure bi-directional shared-memory communication when the program is started on

the .NET side. See the section "Bi-directional shared-memory communications," below, for more information.

- specifyRemotingConfiguration(JavaScheme *remoteProtocol*, String *jvmLocation*,
  String *jnbcoreLocation*, String *bcelLocation*, String *classpath*, String *javaEntryLocation*)

  is a convenience version of the previous version of specifyRemotingConfiguration(), used when bi-directional shared-memory communication (started from the .NET side) is to be used and there are no additional JVM options to be supplied. It is equivalent to calling

  specifyRemotingConfiguration(*remoteProtocol*, *jvmLocation*, *jnbcoreLocation*,
  bcelLocation, *classpath*, new String[0], *javaEntryLocation*);

  See the section "Bi-directional shared-memory communications," below, for more information.

Note that a number of convenience versions of specifyRemotingConfiguration() that were available up through JNBridgePro 10.0 have been removed starting in 10.1.

A call to specifyRemotingConfiguration() must be made before any access of a proxy object in order for it to have an effect. If the call to specifyRemotingConfiguration() is made after an access of a proxy object (for example, a call to a constructor or an access of one of its members), a RemotingException will be thrown.

If a call to specifyRemotingConfiguration() is made, it will override any communications configuration information in the application configuration file or in jnbproxy.config.

.NET-side configuration with .NET Core

One of the biggest differences between traditional JNBridgePro and JNBridgePro for .NET Core is that the .NET Core version uses JSON to specify JNBridgePro configuration. This is because .NET Core applications do not use app.config files. If configured using configuration files rather than programmatically, an application using JNBridgePro for .NET Core must have a file jnbridgeConfig.json, with the following form:

```
{
  "dotNetToJavaConfig": {
    "scheme": "jtcp or sharedmem",
    "host": "remote host name or IP address",
    "port": remotePortNumber,
    "useSSL": "true or false",
    "jvmOptions": [ "list", "of", "jvm options" ],
    "jvm": "path to jvm.dll if using shared memory",
    "jvm32": "path to 32-bit jvm.dll if using shared memory",
    "jvm64": "path to 64-bit jvm.dll if using shared memory",
    "jnbcore": "path to jnbcore.jar",
    "bcel": "path to bcel-6.n.m.jar",
    "classpath": "semicolon-separated classpath"
  },
  "javaToDotNetConfig": {
    "scheme": "jtcp",
    "port": "portNumber",
    "useSSL": "true or false",
    "certificateLocation": "path to SSL certificate",
    "certificatePassword": "password of SSL certificate",
```

```
    "useIPv6": "true or false"
    "useClassWhiteList": "true or false",
    "classWhiteListFile": "path to class whitelist file",
  },
  "assemblyList": [ "list", "of", "assembly paths"],
  "tcpNoDelay": "true or false",
  "licenseLocation": {
    "host": "host",
    "port": "portNumber",
    "directory": "C:\\Program Files (x86)\\JNBridge\\JNBridgePro v12.0"
  },
  "javaSideDeclarations": [
    {
      "javaSideName": "second_PrimaryURL",
      "javaSideURL": "jtcp://localhost:8090/JNBDispatcher"
    },
    {
      "javaSideName": "js2",
      "javaSideURL": "url2"
    }
  ]
}
```

The elements above correspond directly to elements in the traditional JNBridgePro app.config file. Clearly, not all elements are necessary, and some sets of elements are mutually exclusive. See the appropriate sections of the *JNBridgePro Users' Guide* for more information on how these elements are used.

The file jnbridgeConfig.json must be in the same folder as the copy of jnbshare.dll being used by your application.

Note that, while the JSON specification does not allow for comments (a serious oversight, in our opinion), Microsoft's implementation does support comments. Everything from a '//' sequence to the end of a line is considered a comment and ignored.

Programmatic configurating using the JNBRemotingConfiguration.specifyRemotingConfiguration() APIs is also available and works the same way as it does in "traditional" JNBridgePro.

Example: Java-to-.NET Core projects using TCP/binary communications

```
{
  "javaToDotNetConfig": {
    "scheme": "jtcp",
    "port": "portNumber",
    "useSSL": "true or false",
    "certificateLocation": "path to SSL certificate",
    "certificatePassword": "password of SSL certificate",
    "useIPv6": "true or false"
    "useClassWhiteList": "true or false",
    "classWhiteListFile": "path to class whitelist file",
  },
  "assemblyList": [ "list", "of", "assembly paths"],
  "tcpNoDelay": "true or false",   // optional
```

```
    "licenseLocation": {      // use host/port, or directory, but not both
      "host": "host",
      "port": "portNumber",
      "directory": "C:\\Program Files (x86)\\JNBridge\\JNBridgePro v12.0"
    }
  }
```

Example: Java-to-.NET Core projects using shared memory communications

When using shared memory in Java-to-.NET Core projects, you still must have a jnbridgeConfig.json file, but it will only include the location of the license file or the host and port of the license server:

```
{
  "licenseLocation": {  // use host/port or directory, but not both
    //"host": "host",
    //"port": "port",
    "directory": "C:\\Program Files (x86)\\JNBridge\\JNBridgePro v12.0"
  }
}
```

jnbcore.properties file

In order to configure Java-side clients for Java-to-.NET calls, `jnbcore.properties` should contain lines similar to the following:

```
dotNetSide.serverType=tcp
dotNetSide.host=localhost
dotNetSide.port=8086
dotNetSide.useSSL=false
```

The `dotNetSide.*` properties tell the Java-side client how to find and communicate with the corresponding .NET-side. The `dotNetSide.serverType` property can be `tcp` or `sharedmem`, depending on whether the tcp/binary or shared memory communications mechanisms, respectively are used. The `dotNetSide.host` property contains the host name or IP address of the host containing the .NET side. If the .NET side is on the same machine as the Java side, the `dotNetSide.host` property can be localhost, as above. The `dotNetSide.port` property contains the port on which the corresponding .NET side will be listening for requests from the Java side. The `dotNetSide.useSSL` property indicates whether or not the Java-side client should use SSL (secure sockets layer) for secure communications. It may be `true` or `false`; if it is omitted, the default value of `false` will be used. If the server type is tcp, any other `dotNetSide.*` properties in `jnbcore.properties` will be ignored. If the server type is sharedmem, the host, port, and useSSL properties are ignored, and a new set of properties, described in the next section, is used.

The `dotNetSide.*` properties in `jnbcore.properties` must agree with the configuration of the corresponding .NET-side server or the two sides will not be able to communicate with each other.

When `dotNetSide.useSSL` is `true`, there are additional SSL-related properties that must be supplied. See "Secure communications using SSL (Secure Sockets Layer)" for more information.

Configuration of shared-memory communication: If using shared-memory communication in a Java-to-.NET project, it is necessary to add the following entry to the dotNetSide.* properties:

```
dotNetSide.javaEntry=full path to JNBJavaEntry_x86.dll
```

or

```
dotNetSide.javaEntry=full path to JNBJavaEntry_x64.dll
```

depending on whether the application will be running on a 32-bit or a 64-bit Java Runtime Environment (JRE).

If your application should be able to run, unchanged, on both 32-bit and 64-bit JREs, set the dotNetSide.javaEntry property as follows:

```
dotNetSide.javaEntry=full path to folder containing both
     JNBJavaEntry_x86.dll and JNBJavaEntry_x64.dll
```

and the application will load the correct version of JNBJavaEntry depending on whether it is running as a 32-bit or 64-bit process.

This tells the Java side where to find the code that allows it to link with the .NET side.

In addition, if one will be accessing .NET classes other than those in mscorlib.dll from .NET, one must specify the assemblies in which those classes appear so that they will be loaded in the .NET side. This is done as follows:

```
dotNetSide.assemblyList.1=name or path of assembly to be loaded
dotNetSide.assemblyList.2=name or path of assembly to be loaded
…
```

There should be one entry, each with a unique sequence number, for each assembly to be loaded. Either the full path to the assembly must be supplied, or, if the assembly is in the GAC, the assembly's full name, including version number, culture, and public key token, must be supplied.

Alternatively, if the list of assemblies to be loaded is supplied in .NET application configuration file (see the section "Configuring the .NET side," above), the entry

```
dotNetSide.appConfig=full path to application configuration file
```

can be used to specify the configuration file containing the assembly load information.

Both `dotNetSide.assemblyList.*` and `dotNetSide.appConfig` entries can be used in the same jnbcore.properties file. If they are both used, all the specified assemblies will be loaded into the .NET side.

If the .NET-side code being accessed from .NET requires configuration through an application configuration file, the `dotNetSide.appConfig` entry in jnbcore.properties should be used to specify the location of the application configuration file.

Note: In all cases in which file paths are supplied in the properties file (for example, in dotNetSide.javaEntry, dotNetSide.assemblyList.*, and dotNetSide.appConfig, above), one must use forward slashes (/) or double backslashes (\\), but not single backslashes, in the file paths. For example, C:\a\b\c is not a valid file path in the properties file, but C:/a/b/c and C:\\a\\b\\c are both valid.

In certain situations, it may be necessary to specify the .NET-side application base folder, so that the .NET-side applications can find assemblies to be dynamically loaded. If this is necessary, use the `dotNetSide.appBase` property. See the section "Specifying the .NET-side application base folder," below.

Configuring the Java side when using .NET Core and shared memory

Java-side configuration works the same way as in "traditional" JNBridgePro, with one additional property that may be configured:

dotNetSide.coreClrPath=*path_to_folder_containing_coreClr.dll_or_libcoreclr.so*

Use this in Java-to-.NET Core projects that use shared memory communication. The property specifies the folder containing coreClr.dll (on Windows) or libcoreclr.so (on Linux). This is the file containing the .NET Core CLR (Common Language Runtime) implementation. For example, on Windows it could be:

dotNetSide.coreClrPath=C:/Program Files/dotnet/shared/Microsoft.NETCore.App/3.0.0

As with other paths in the Java-side configuration, use forward-slashes ('/') rather than backslashes ('\').

Starting a .NET side (CLR) for proxy use

Note: The instructions in this section only apply when TCP/binary communications is used. When using shared-memory communications, it is not necessary to explicitly start the .NET side; it is automatically started when main Java process is started.

In order to create a .NET side whose classes can be called from Java, one must start up a .NET-side server. There are several ways to do this. The simplest way is to use the standalone .NET-side server supplied in JNBDotNetSide.exe. JNBDotNetSide.exe receives its configuration, including the protocol it uses, the port it listens on, and the assemblies it can access, through its application configuration file JNBDotNetSide.exe.config. See "Configuring the .NET side: Configuring communications through the application configuration file," above, for more information.

Note: jnbdotnetside.exe runs as a 64-bit process on 64-bit systems, and as a 32-bit process on 32-bit systems, while jnbdotnetside32.exe always runs as a 32-bit process. Use jnbdotnetside32.exe when your .NET side contains x86-targeted DLLs, particularly when you are running the .NET side on a 64-bit system. To use jnbdotnetside32.exe, your folder (or GAC) needs to include jnbdotnetside32.exe, jnbdotnetside32.exe.config, jnbdotnetside.exe, jnbshare.dll, and, optionally, jnbsharedmem_x86.dll. Place the .NET-side configuration information in jnbdotnetside32.exe.config, not in jnbdotnetside.exe.config.

One can also start the .NET-side server programmatically from .NET code by calling

com.jnbridge.jnbcore.DotNetSide.startDotNetSide(string[] *assemblyList*) or
com.jnbridge.jnbcore.DotNetSide.startDotNetSide().

assemblyList is a an array of strings representing the full or relative paths of the assemblies in which the .NET side should search for .NET types being accessed from the Java side. If startDotNetSide() is called without parameters, the .NET-side server is configured through the current application configuration file. (See "Configuring the .NET side: Configuring communications through the application configuration file," above.) If an assembly is in the Global Assembly Cache (GAC), it can be specified using its fully-qualified name. For example, the .NET 2.0 version of the assembly System.Windows.Forms.dll may be specified as

"System.Windows.Forms, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089"

To access all the assemblies in a folder, simply supply the full path of the folder.

Types defined in the assembly mscorlib.dll are always searched and these assemblies need not be listed in the *assemblyList*. It should also be noted that it is not necessary to include references to the assemblies in *assemblyList* in the project calling DotNetSide.startDotNetSide() unless the types are directly referenced from that project.

Note: If a .NET class is accessed from Java that is in an assembly that is not in the assembly list (or is not in mscorlib.dll), a com.jnbridge.jnbcore.DotNetClassNotFoundException will be thrown back to the Java side. In some cases this exception may be wrapped inside a java.lang.ExceptionInInitializerError.

If an assembly in *assemblyList* cannot be found, or its fully-qualified assembly name is incorrect, a System.IO.FileNotFoundException will be thrown.

Once DotNetSide.startDotNetSide() is called, or JNBDotNetSide.exe is launched, a .NET-side server is started and is available to accept requests from Java sides.

The .NET-side server can be stopped by halting JNBDotNetSide.exe, by ending the application that called DotNetSide.startDotNetSide(),programmatically by calling

com.jnbridge.jnbcore.DotNetSide.stopDotNetSide().

Any project containing a call to DotNetSide.startDotNetSide() or DotNetSide.stopDotNetSide() must contain a reference to jnbshare.dll.

Starting and configuring a Java-side client

A Java-side program that accesses .NET classes must first execute

com.jnbridge.jnbcore.DotNetSide.init(String *propertiesFilePath*)

where *propertiesFilePath* is the full path of the jnbcore.properties file containing the dotNetSide.* configuration properties described in the section on Java-side configuration above. The DotNetSide class is located in jnbcore.jar, and jnbcore.jar must therefore be in the classpath of the Java-side client program during compilation and execution.

Alternatively, the Java-side program can first execute

com.jnbridge.jnbcore.DotNetSide.init(java.util.Properties *propertiesTable*)

where *propertiesTable* is a properties table object containing the dotNetSide.* configuration properties described in the section on Java-side configuration above. The DotNetSide class is located in jnbcore.jar, and jnbcore.jar must therefore be in the classpath of the Java-side client program during compilation and execution.

If DotNetSide.init() is not called before the first Java-side proxy is accessed by the Java-side client program, an exception will be thrown.

A Java-side client program must have jnbcore.jar and bcel-*6.n.m*.jar in its classpath.

Secure communications using SSL (Secure Sockets Layer)

SSL is a security feature in JNBridgePro. For a unified discussion of security in JNBridgePro, see the section "JNBridgePro and security."

JNBridgePro supports secure cross-platform communications using SSL (secure sockets layer). SSL is turned on by default when using TCP/binary communications, and includes server authentication, client authentication, and encryption. Server authentication ensures that your Java sides are communicating with a particular .NET side, and prevents other servers from pretending to be your .NET side. Client authentication ensures that only specifically permitted Java sides can access the .NET side, but no others. Encryption ensures that nobody else can eavesdrop on the TCP/binary communications between the .NET and Java sides.

SSL is turned off by default, since configuring it is a bit more complex than the other security features. You can leave it off, but you will be opening yourself to a number of potential attacks, including the ability of any client to execute any code on your Java side's machine. (IP and class whitelisting can mitigate these hazards.)

SSL is not used in shared memory communication. With shared memory, the .NET and Java sides run in the same process, and communication is done using shared data structures, and is inherently secure.

To implement SSL, you need the following certificates for each .NET and Java side.

Each .NET side requires an X.509 certificate, typically contained in a .cer file, and a PKCS#12 or PFX file containing the public key (also in the corresponding .cer file) and the private key, and is typically contained in a .p12 or .pfx file. The .p12 and .pfx files typically have an associated password. In the context of Java-to-.NET interop, these will be known as *server certificates*. The *common name* (CN) field of the server certificate should be the name of the server on which the .NET side resides.

Each Java side also requires two files: an X.509 certificate, typically contained in a .cer file, and a PKCS#12 or PFX file containing the public key (also in the corresponding .cer file) and the private key, and is typically contained in a .p12 or .pfx file. The .p12 and .pfx files typically have an associated password. In the context of Java-to-.NET communications, these .cer and .p12/.pfx files are known as *client certificates*.

On the .NET-side machine, install the .p12/.pfx server certificate file in the windows certificate store. Do so by double-clicking on the file in Windows Explorer or by right-clicking and selecting Install certificate…, and following the sinstructions. Also, leave the server .cer certificate file on the .NET-side's disk drive – you will be supplying the path to the .cer file as part of the configuration. After installing the .p12/.pfx, you can remove it from the .NET-side machine's file system, although you should keep a copy in a decure place.

Also on the .NET-side machine, install each authorized Java side's client certificate (the .cer file, not the corresponding .p12/.pfx file) in that machine's Windows certificate store. Do so by double-clicking on the file in Windows Explorer or by right-clicking and selecting Install certificate…, and following the instructions. *(If the certificate is self-signed, it must be imported into either the* ***Trusted Root Certification Authorities*** or the ***Third-Party Root Certification Authorities*** *stores. When the installation askes you where the certificate should be installed, you must select one of those options.)*

On the Java side, install the client certificate .p12/.pfx file (the secure key pair, not the .cer file) in a Java *keystore* (.jks) file. Note that there are many tools to create and manage certificates and .jks files, including the keytool that comes with the JDK, and the free, open source, GUI-based KeyStore Explorer. All should work, although we find the KeyStore Explorer particularly easy to use.

Also on the Java-side machine, install each authorized .NET side's client certificate (the .cer file, not the corresponding .p12/.pfx file) in a different .jks file known as the *truststore*. Note that the keystore and the truststore should have passwords associated with each.

Configure the Java side as follows by supplying the following properties, either in the Java-side properties file, or programmatically:

- javaSide.useSSL=true # if this is left out, the value is false by default
- javaSide.keyStore=*path_to_keystore_.jks_file* # the path to the file itself, not to the folder
- javaSide.keyStorePassword=*keystore_password*
- javaSide.trustStore=*path_to_truststore_.jks_file*
- javaSide.trustStorePassword=*truststore_password*

Note that the keystore and truststore passwords should probably be configured programmatically from the contents of a secure file, or from values input by users. Placing them inside the .properties file, where they can be read by anyone, is probably a bad idea.

On the .NET side, if you are configuring the .NET side using an app.config or web.config file, use the following <dotNetToJavaConfig> element:

```
<javaToDotNetConfig
    scheme="jtcp"
    host="hostname"
    port="port number"
    useSSL="true"
    serverCertificateLocation="path to .p12 or .pfx file"
    serverCertificatePassword="path to .cer file"
/>
```

serverCertificateLocation is assigned the path to the server certificate – the .p12 or .pfx file (which includes the private key). It must be a path to the file itself, not to the containing folder. *serverCertificatePassword* is the password of the server certificate file. SSL can also be configured programmatically on the .NET side. For more information, please see the section on the programmatic configuration APIs.

SSL can be turned off. On the Java side, set the property javaSide.useSSL=false. On the .NET side, use the app.config element

```
<javaToDotNetConfig
    scheme="jtcp"
    host="hostname"
    port="port number"
    useSSL="false"
/>
```

If SSL is turned off, it must be turned off in all .NET sides and Java sides that participate in the application. Again, note that if SSL is turned off, you open yourself to other attacks, and should mitigate them by employing other security mechanisms offered by JNBridgePro (including IP and class whitelisting), making sure that your firewall is properly configured, or using shared memory instead.

Class whitelisting

Class whitelisting is a security feature in JNBridgePro. For a unified discussion of security in JNBridgePro, see the section "JNBridgePro and security."

Starting with JNBridgePro 10.1, it is possible to specify which classes' APIs are available for cross-platform access, through the class whitelisting feature. Without whitelisting, it is possible for unauthorized clients to access APIs on the .NET-side machine. It is even possible for them to execute arbitrary code on the Java side machine through use of the Process API, and it is even possible for this to happen from clients that are not using the JNBridgePro proxies and runtime components. Client whitelisting mitigates this problem by only allowing specific APIs to be accessed remotely.

When class whitelisting is turned on (which it is by default), a list of classes is read from a text file known as the class whitelist file. The file contains one class per line, and should contain no additional commas or other delimiters. (Whitespace surrounding the class names is allowed.) The following is an example of the contents of a class whitelist file:

```
pkg1.Class1
pkg2.Class2
pkg3.Class3
```

If a class is whitelisted, then the .NET side will allow calls to instance members of objects of that class, and it will allow calls to static members and constructors of that class.

In addition, if an interface is whitelisted, then any other class that implements the interface is considered whitelisted. Similarly, if a class is whitelisted, then any other class that inherits from the whitelisted class is also considered whitelisted. Additionally, only raw generic classes (e.g, MyGenericClass`1), can be placed in the whitelist, and this means that all instantiations of that raw generic class are whitelisted. Finally, if a class is whitelisted, then all of its nested classes are considered whitelisted.

The only exception to the above is System.Object. It is whitelisted by default, so that methods declared in the Object class are accessible, but classes that inherit from Object (that is, all classes) are not automatically whitelisted – otherwise, all classes would be whitelisted.

In addition to Object, other classes that are automatically whitelisted are System.Type, System.String, System.Int32, System.Int16, System.Int64, System.Single, System.Double, System.Boolean, System.Byte, System.SByte, System.Char, System.UInt16, System.UInt32, and System.UInt64.

All other classes must be explicitly whitelisted, or inherit from or implement classes or interfaces that have been explicitly whitelisted.

If a class has not been whitelisted, then any access to that class, or to objects of that class, from a .NET client or any other client over a TCP/binary connection, will cause an exception to be thrown.

The class whitelist is specified through the following Java-side property:

```
<javaToDotNetConfig scheme="jtcp" port="portNumber"
       useClassWhiteList="true"
       classWhiteListFile="path to class whitelist file" />
```

```
The useClassWhiteList element is optional. If it is omitted, the default value
true is used.
```

For example,

```
<javaToDotNetConfig scheme="jtcp" port="portNumber"
       useClassWhiteList="true"
       classWhiteListFile=" C:/Documents/classWhiteList.txt" />
```

It is also possible to turn off class whitelisting through the following Java-side property:

```
<javaToDotNetConfig scheme="jtcp" port="portNumber"
       useClassWhiteList="false" />
```

Class whitelisting can also be controlled through the JNBRemotingConfiguration.specifyRemotingConfiguration() programmatic configuration API.

We only recommend turning off whitelisting during the early stages of development. During testing and deployment, we strongly recommend using whitelisting.

Whitelisting is only offered when TCP/binary communications is used. It is not offered when shared memory is used, since remote clients will not be accessing the Java side, and it is therefore not necessary.

Whitelisting Java clients

IP whitelisting is a security feature in JNBridgePro. For a unified discussion of security in JNBridgePro, see the section "JNBridgePro and security."

By default, for security reasons, a JNBridgePro .NET side will accept requests from .NET sides on the same machine. It may be desirable to specify a whitelist of additional hosts from which a .NET side will accept Java requests. To specify such a whitelist, add the following element to the .NET-side configuration:

<javaToDotNetConfig
scheme="jtcp"
port=*port_responding_to_requests*
… other specification elements …
ipWhitelist="*semicolon-separated list of IP addresses*"
/>

Note that case matters when specifying the ipWhitelist element.

The IP addresses may be in IPv4 ('.'-separated) or IPv6 (":"-separated) style. "*" may be used as a wildcard in place of any element. "*" or "::" may be used as a wildcard to match any IP address.

If the ipWhilelist element is not specified, it is equivalent to having specified

ipWhitelist="127.0.0.1;::1"

that is, it will accept requests from any client on the same machine.

As an example,

ipWhitelist="1.2.3.4"

will cause the .NET side to only accept requests from the machine whose IP address is 1.2.3.4. All other requests will be rejected, and any Java side making a proxy call to that .NET side from any other machine will see an exception.

As another example,

ipWhitelist="127.0.0.1;10.1.2.* "

will cause the .NET side to only accept Java requests from the same host, or from any host in the range 10.1.2.0 to 10.1.2.255, and reject all others.

Specifying the .NET-side application base folder

In most cases, the .NET assemblies are loaded through the `<assemblyList>` element of the .NET-side configuration file or `dotNetSide.assemblyList` properties of the Java-side properties file.

However, in some cases, the .NET-side assemblies will load other assemblies. They typically find these assemblies by searching the folders below the "application base" folder, which is generally the folder containing the running .exe file. When shared memory is used, the application base folder is typically the folder containing java.exe, which often presents a problem, since the containing folder (the bin folder in the Java runtime environment installation) is not the folder that contains the .NET assemblies. In situations like this, where the .NET-side attempts to load assemblies, looks in the wrong place, cannot find them, and throws an exception, it is possible to specify the folder to use as the application base of the .NET side for the purposes of searching for assemblies to load. To specify the application base, use the property

`dotNetSide.appBase=`*path to application base folder*

Assemblies used by JNBridgePro can also be placed in the application base folder instead of being placed in the Global Assembly Cache (GAC).

Permissions to run a .NET side

Any user accessing proxies generated by JNBridgePro must "Read" and "Write" access to the folder `<System-drive>:\Documents and Settings\All Users\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.

Tuning 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 the typical JNBridgePro-generated network packet is small, and in some cases this may lead to network congestion and degraded performance. If you encounter network performance degradation, you can turn off this NoDelay option so that packets are aggregated before they go out. In some cases this may improve network performance. Typically, if calls or returns contain a large amount of data (for example, if you are passing large arrays, or large by-value objects), 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 your .NET application's configuration file. That is, if your 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>
   <sectionGroup name="jnbridge">
   <section name="dotNetToJavaConfig"
             type="System.Configuration.SingleTagSectionHandler" />
   <section name="javaToDotNetConfig"
             type="System.Configuration.SingleTagSectionHandler" />
   <section name="tcpNoDelay"
             type="System.Configuration.SingleTagSectionHandler" />
   <section name="javaSideDeclarations"
             type="System.Configuration.NameValueSectionHandler" />
   <section name="assemblyList"
             type="com.jnbridge.jnbcore.AssemblyListHandler, JNBShare,
Version=12.10.0.0, Culture=neutral, PublicKeyToken=b18a44fb28aea122" />
   </sectionGroup>
   <jnbridge>
```

```
      <tcpNoDelay noDelay="false" />
 </jnbridge>
</configuration>
```

Note that for historical reasons, for the 4.8-targeted version of JNBridgePro being used, the "assemblyList" definition above should refer to "JNBShare, Version=12.10.0.0," not "12.0.0.0".

To turn the NoDelay option off in the Java-to-.NET direction, add the following line to your jnbcore.properties file

```
javaSide.nodelay=false
```

Deploying JNBridgePro as part of another application

While JNBridgePro installer can be run in "deployment configuration," this is not necessary when deploying an application that uses JNBridgePro. All that is necessary is to deploy the application including the proxy dll and the various JNBridgePro components, including jnbdotnetside.exe (with configuration file), jnbshare.dll, jnbsharedmem (jnbsharedmem_x86.dll, jnbsharedmem_x64.dll, or both), jnbjavaentry (jnbjavaentry_x86.dll, jnbjavaentry_x64.dll, or both), jnbjavaentry2 (jnbjavaentry2_x86.dll, jnbjavaentry_x64.dll, or both), jnbcore.jar, and bcel-*6.n.m*.jar. In addition, you must deploy either jnbauth_x86.dll or jnbauth_x64.dll, depending if your application is a 32-bit or 64-bit application. If your application is expected to run without change under both a 32-bit and a 64-bit JRE, you should deploy both jnbauth_x86.dll and jnbauth_x64.dll. jnbauth_x86.dll and jnbauth_x64.dll, when deployed, must be placed in the same folder as the application's jnbshare.dll. In addition, if the application can run under both 32-bit and 64-bit JREs and is using shared memory, make sure to include both the 32-bit and 64-bit versions of jnbsharedmem, jnbjavaentry, and jnbjavaentry2, and set the dotNetSide.javaEntry property in the configuration to point to the folder *containing* both jnbjavaentry_x86.dll and jnbjavaentry_x64.dll.

Note that the licensing activities discussed below are discussed in detail in the section "Licensing," above.

If license activation is required, RegistrationTool.exe must also be deployed to the folder containing jnbshare.dll. In this case, perform license activation using the registration tool. If activation is not required and a license file is available, the file should be deployed into the application executable's folder, or into the JNBridgePro installation folder if the installer was run. If you have an activation key for an evaluation license, you can use it to obtain a 30-day evaluation license, then obtain a permanent license later. You can also run the registration tool from a script, using its command-line interface. Finally, you can add information to the application configuration file indicating the location of the license file, or of a license manager, if one is being used.

Using JNBridgePro with .NET Framework 3.5 and earlier

JNBridgePro will work with.NET Frameworks 4.0 and later. JNBridgePro 12.0 will *not* work with .NET Frameworks 3.5 and earlier.

Using generated proxies

Development

Once proxies have been generated, you can write Java code that uses the proxies to call the corresponding .NET classes. In order to access the .NET classes during development, you must generate jar file containing proxy classes as described above. In addition, the jar file `jnbcore.jar` must be installed on the development machine. In the development environment, add the proxy jar file (in the example above, `jnbproxies.jar`) and `jnbcore.jar` to the build classpath. You can then call the .NET classes by writing calls to the proxies. The proxies have the same name as their corresponding .NET class, and reside in a package identical to the namespace in which the corresponding .NET class resides. In the example above, you can make calls to the .NET class Customer by making calls to the Java proxy class Customer in the package com.jnbridge.samples.

For example, to create an instance of the .NET class Customer, you can write the following code, using Java:

```
import com.jnbridge.samples.*;
…
Customer c = new Customer();
```

you can then call the proxy object as if you were directly calling the Java object:

```
int i = c.getLastOrder();
```

Static calls are also possible:

```
int j = Customer.getLastCustomerID();
```

Fields may be accessed through getter and setter methods:

```
string s = c.Get_name();
c.Set_name("new customer name");
```

Constructor calls, parameter passing, and return values all work as though the .NET classes were being called directly:

```
Person p = c.getContact();
c.setContact(new Person("John Q. Public"));
```

Thrown exceptions are also caught as if you were directly catching the Java exceptions:

```
try
{
        c.addOrder("an invalid order");
}
catch (InvalidOrderException)
{
        // handle the exception here
}
```

(For a full discussion of the way JNBridgePro maps .NET classes to Java proxies, including handling of interfaces and abstract classes, see *Mapping Java entities to .NET*, below.)

You not only have the ability to use JNBridgePro to call .NET classes and objects, you can also inherit from .NET classes and objects. For example, assuming the example given above, you can create a Java class InactiveCustomer by inheriting from Customer as follows:

```
import com.jnbridge.samples.*;
…
public class InactiveCustomer extends Customer
{
        dateTime dateInactivated;
        public boolean customerStatus();
}
```

In addition to the new field dateInactivated and the overriden method customerStatus(), all the functions and fields available in Customer are also available in InactiveCustomer, and constructing a new InactiveCustomer will cause invocation of Customer's constructor, as expected.

You can also pass an instance of InactiveCustomer wherever an instance of Customer is expected:

```
InvalidCustomer ic = new InvalidCustomer();
…
Customer.registerCustomer(ic);
```

There is one restriction to the use of inheritance. The Java subclasses to the Java classes are not recognized by the .NET-side, so that in the example above, if the .NET static method Customer.registerCustomer() looks like the following:

```
static public void registerCustomer(Customer c)
{
        bool b = c.customerStatus();
}
```

the customerStatus() method defined in class Customer will be called, not the overriding method in class InvalidCustomer. This restriction will be removed in a future version.

Garbage collection and object lifecycle management

In JNBridgePro, when proxy objects are no longer reachable and are garbage collected, the finalization routine will notify the corresponding server that the object no longer has an external reference and may itself be ready for garbage collection. For example, if a Java proxy object is garbage collected, it will notify the .NET side that the corresponding .NET object is no longer needed on the Java side. Generally, nothing more needs to be done, and object lifecycle management is completely transparent. However, there are some situations where it is desirable to have finer control over object lifecycle management. For example, Java garbage collection is typically triggered by memory usage exceeding certain threshold levels. Consider a situation where a Java proxy object represents a very large .NET object, or the .NET object it represents is at the root of a very large set of .NET objects. Even if the proxy object is no longer reachable, Java garbage collection may not be triggered because there is little memory pressure on the Java side, the .NET-side objects are not released, and the .NET side may run out of memory even through the .NET-side objects are no longer being used.

JNBridgePro offers a mechanism for releasing .NET objects before their Java proxies are garbage collected. All Java proxies implement a dotNetDispose() method, which releases the corresponding .NET object so that it becomes eligible for garbage collection. If a Java proxy is accessed after its dotNetDispose() has been called, a com.jnbridge.jnbcore.DotNetObjectDisposedException will be thrown.

Warning: Lifecycle management of Java proxies and the corresponding .NET objects works similarly to the lifecycle management of .NET proxies and their corresponding Java objects, although Java garbage collection is not as strictly specified as .NET garbage collection. In particular, Java

platforms will by default not call the finalizers of any Java objects that are still active upon termination (unlike .NET). (It is possible to force the JVM to call the finalizers of all objects remaining on the heap upon exit from the application by calling System.runFinalizersOnExit(true) or Runtime.runFinalizersOnExit(true) somewhere in the application. However, these methods have been deprecated as of JDK 1.2 and are considered unsafe.) Consequently, it is advisable to call the method

```
static void com.jnbridge.jnbcore.DotNetSide.disposeAll()
```

before a Java-side client program terminates in order to release all unreleased .NET objects referenced by proxies. It is also strongly recommended that dotNetDispose() be called on Java-side proxies as soon as they are no longer needed, since it is not guaranteed that the Java platform will finalize the Java-side proxy in a timely manner upon garbage collection and release the corresponding .NET objects, or even that the finalizer will be called at all.

Similarly, when calling multiple .NET sides from Java, one can call the method

```
static void com.jnbridge.jnbcore.DotNetSide.disposeAll(String dotNetSideName)
```

to dispose of all proxies and release underlying .NET objects associated with a single, named .NET side.

Running the application

To run the application once it has been written and built, first configure the .NET- and Java-sides as described in the section *System configuration for proxy use*, above. Then, start the Java-side as described in the section *Starting a standalone JVM for proxy use*, above. Finally, start the .NET-side application.

Data compression

The tcp/binary communications mechanism in JNBridgePro now compresses binary messages over a certain size before exchanging them between the .NET and Java sides. This process is transparent to the user, and ordinarily nothing needs to be done about it. Please note that to prevent the Java side from returning compressed messages, the javaSide.noCompress property can be set to "true". If it is missing, or set to anything else, the Java side will return compressed messages.

Embedding .NET GUI elements inside Java GUI applications

It is possible to embed .NET GUI elements, that is, Windows Forms controls or Windows Presentation Foundation (WPF) controls, inside Java GUI applications. To do so, the proxied .NET GUI element (which must be derived from System.Windows.Forms.Control, if Windows Forms, or System.Windows.Controls.Control, if WPF), must be wrapped inside a com.jnbridge.embedding.DotNetControl object as follows:

```
MyWinFormControl mc = new MyWinFormControl ();      // MyWinFormControl is a proxy of a
                                                    // .NET GUI control
DotNetControl dnc = new DotNetControl (mc);
```

The DotNetControl object is derived from java.awt.Component, and can be used wherever any other AWT component can be used.

If the .NET GUI element will be used inside a Java SWT application, the proxied .NET GUI element must be wrapped inside a com.jnbridge.embedding.DotNetSWTControl, which inherits from org.eclipse.swt.widgets.Composite, and can be used wherever any other SWT Composite can be used:

```
MyWinFormControl mc = new MyWinFormControl();        // MyWinFormControl is a proxy of a
                                                     // .NET GUI control
DotNetSWTControl dnc = new DotNetSWTControl(mc, parent);
                                                     // parent is the parent SWT composite
```

Any listeners on events in the .NET control can be registered directly with the .NET control proxy object using the usual callback mechanism.

Embedding .NET GUI elements inside Java GUIs can only be done when using shared memory communications.

When using DotNetSWTControl, make sure that the folder containing the native SWT libraries (typically, inside the Eclipse installation – eclipse\plugins\org.eclipse.swt.win32_3.1.0\os\win32\x86 – although it may be elsewhere) is inside the java.library.path system property. Use a –D option on the Java command-line to specify the Java library path value (e.g., java –Djava.library.path=*java library path here* …).

For more information, please see the "Embedding .NET in Java GUIs" example that comes with the JNBridgePro distribution.

Important note: When embedding a WPF control inside a Java program, you ***must*** use the code described in the next section, "Embedding multi-threaded .NET controls in Java GUI applications." If you do not, the control will not render properly.

Important note: When embedding a WPF control inside a Java program, your application must reference the assembly jnbwpfembedding.dll, and it must be deployed into the Global Assembly Cache (GAC), or into the folder specified in the dotNetSide.appBase property (if such a property is specified – see "Specifying the .NET-side application base folder").

Resizing embedded .NET controls

The Java (AWT/Swing and SWT) and .NET (Windows Forms and WPF) layout mechanisms are not integrated, so, when a surrounding Java application or control is resized and the embedded .NET control is resized, the resizing must be done explicitly. To support this, the DotNetControl and DotNetSWTControl classes offer the method

```
void setEmbeddedSize(double width, double height)
```

To use setEmbeddedSize(), create and register a resize listener for the wrapper object (that is, for the DotNetControl or DotNetSWTControl object), and inside the body of the listener call setEmbeddedSize() with the new size of the control. The embedded .NET control will be automatically resized. This will work regardless of whether the embedded .NET control is a Windows Forms control or a WPF control.

Embedding multi-threaded .NET controls in Java GUI applications

Some .NET controls perform work in the background, on BackgroundWorker threads and timer threads. When these controls are embedded in Java applications using the JNBridgePro embedding mechanism, it is possible that these background threads may be blocked because messages in the Windows Forms event queue are not being properly consumed. If the control's background threads

are blocked, and work is not being performed, one can assure that the Windows Forms and WPF events are being properly consumed as follows (assuming you are creating an AWT or Swing application – for SWT, please see below):

- First, make sure that the class System.Windows.Forms.Application has been proxied.
- Your Java application's main procedure should contain the following code:

```
public static Boolean mustClose = false:

public static void main(String[] args) {
   // main application code here, including
   // initializing .NET side (call DotNetSide.init()),
   // instantiating and wrapping the .NET control,
   // laying out the GUI and setting the various event listeners,
   // and calling setVisible() on the outermost Frame

   while(!mustClose)
   {
         Application.DoEvents(); // Application is a proxy
   }

}
```

- In addition, the Java application must have a WindowListener with a windowClosing() method that sets the mustClose flag to true, causing the Application.DoEvents() loop to exit. For example, the code could be as follows (assuming f is the outermost Frame):

```
f.addWindowListener(new WindowAdapter() {
   public void windowClosing(WindowEvent evt) {
         Window w = evt.getWindow();
         w.setVisible(false);
         w.dispose();
         // if application hangs on exit, add a delay here
         // try
         // {
         //    Thread.sleep(50); // alter delay value as appropriate
         //} catch (InterruptedException e) { }
         mustClose = true;
   }
});
```

Note that mustClose must be set to true *after* the rest of the windowClosing() method has executed, so that the DoEvents() loop does not terminate until after windowClosing() completes. If the DoEvents() loop exits prematurely, the application may hang before it closes.

Please note that, if you are unsure about whether your embedded control is multithreaded, it is always safe to add the above Application.DoEvents() loop and the associated code.

Embedding in SWT: If you are embedding a multi-threaded .NET control inside an SWT application, you will need to place the calls to Application.DoEvents() inside the SWT read-and-dispatch loop. Before embedding the .NET control, your application should have a piece of code like the following:

```
while (!shell.isDisposed()) {
   if (!display.readAndDispatch()) {
         display.sleep();
```

```
        }
    }
```

To properly handle events in an embedded multi-threaded .NET control, make sure you have proxied System.Windows.Forms.Application, and modify that code as follows.

```
    while (!shell.isDisposed()) {
        if (!display.readAndDispatch()) {
            Application.DoEvents();
            display.sleep();
        }
        else {
            Application.DoEvents();
        }
    }
    Application.Exit();  // when embedding WPF controls
```

When embedding WPF controls inside SWT applications, it may be necessary to place the call to Application.Exit() after the while loop, as shown above. In those situations, if Application.Exit() is not there, the application's process may not terminate when the SWT application is exited.

Specifying the .NET-side apartment threading model when using shared-memory communications

When using shared memory in a Java-to-.NET application, the .NET side is started by default using the multi-threaded apartment (MTA) threading model. (Please see the MSDN documentation for details of the threading apartment model.) Some .NET components, particularly embedded UI controls, may require the single-threaded apartment (STA) model. To specify the apartment threading model used by the .NET side, add the following property to the Java-side properties file, or supply it programmatically when configuring the Java side:

`dotNetSide.apartmentThreadingModel=STA` *or*

`dotNetSide.apartmentThreadingModel=MTA`

If the dotNetSide.apartmentThreadingModel property is omitted, the default of MTA will be used.

Binding an IP address

By default, a JNBridgePro .NET side will accept requests addressed to any IP address for which the host machine will accept requests. For security purposes, it may be desirable to specify that the .NET side only accept requests from one particular address. To specify such a binding, add the following element to the .NET-side configuration:

<javaToDotNetConfig
    scheme="jtcp"
    port=*port_responding_to_requests*
    … other specification elements …
    hostIP="*IP address bound to .NET side*"
/>

The IP address must be a properly formatted IPv4 or IPv6 address. It may also be "*" or "::", representing any IP address associated with the host machine. Omitting the hostIP element is equivalent to setting

hostIP="*"

Interacting with Multiple .NET-sides

It is possible to have a Java-side on one machine communicate with multiple .NET-sides. These .NET-sides may be on multiple machines, or there may be several CLRs on one machine, each running its own Java-side. A .NET-side may use a different communication protocol in communication with each Java-side; there is no requirement that the same protocol be used in all cases.

Each .NET-side is configured and started in the usual manner. If multiple .NET-sides are running on one machine, then each .NET-side must be configured to listen on a different port, which implies that each .NET-side must have its own configuration file.

A Java-side always has exactly one *default .NET-side* with which it communicates. The location (host name and port) of the default Java-side, and the protocol used to communicate with it, are contained in the main properties file that is referenced in a call to DotNetSide.init(*propertiesFilePath*), or specified programmatically through a call to DotNetSide.init(*propertiesObject*), either of which must be present. Additional .NET-sides can be declared in the in the Java program through calls using the API below. At any given time, there is exactly one *active* .NET-side; when the program starts, the default .NET-side is always the active .NET-side, but the active .NET-side can be changed at any time to be one of the other declared .NET-sides or back to the default .NET-side.

When a proxy is instantiated, that instance will always communicate with whatever .NET-side was active at the time of its creation, even if the active .NET-side is subsequently changed. Thus, calls to any instance methods or fields in that proxy will always go to the .NET-side that was active at the time of the proxy creation. In addition, any new proxies that are returned by instance methods of that proxy, or by accesses to fields, will also subsequently communicate with the same .NET side as that which was bound to the proxy whose method or field returned the new proxy.

In contrast, all calls to static methods and fields are sent to the .NET-side that is active at the time of the call.

In all cases, it is the responsibility of the user to make sure that the desired .NET class is present on the target .NET-side. If it is not, a MissingTypeException will be thrown.

Also, it is not permissible to pass a reference to a .NET object on one .NET-side as a parameter to a .NET method residing on a different .NET-side. The results of such an operation are undefined: an exception might or might not be thrown, and the results will most likely be incorrect. It is the responsibility of the user to make sure that this does not happen.

The JNBridgePro class com.jnbridge.jnbcore.DotNetSide provides an API for declaring new .NET-sides and for changing the active .NET-side.

- **static void addServer(String name, String configFile)**
  static void addServer(String name, Properties configProperties)
  Declare a new .NET-side with which the current Java side can communicate. *name* is the name of the new .NET-side and must not have been previously used, or a DotNetSideException will be thrown. *configFile* is the path to the properties file that defines how the Java side will communicate with the new .NET side. *configProperties* is the equivalent properties definition as a Properties object, and is used for programmatic configuration. The contents of the properties file or object are as defined in the section "jnbcore.properties," above.
  Note: the name "default" is reserved for the default .NET-side, and may not be used as a value for the name argument in addServer().

- **static void setServer(String name)**
  static void setServer(String name, boolean allThreads)
  Set the named .NET-side to be the new current .NET-side. A .NET-side of the specified name must have been previously declared using addServer(), or a DotNetSideException will be thrown. The *allThreads* parameter indicates whether the active .NET side should be set to the specified .NET side for all of the program's threads (if *allThreads* is **true**), or if it should be set for just the current thread (if *allThreads* is **false**). setServer(*name*) is a convenience method equivalent to calling setServer(*name*, **true**).

- **static void resetDefaultServer()**
  static void resetDefaultServer(boolean allThreads)
  Set the default .NET-side to be the new current .NET-side. The *allThreads* parameter indicates whether the active .NET side should be set to "default" for all of the program's threads (if *allThreads* is **true**), or if it should be set for just the current thread (if *allThreads* is **false**). resetDefaultServer() is a convenience method equivalent to calling resetDefaultServer(**true**).

- **static String currentServer()**
  Returns the name of the currently active .NET side.

- **static String boundServer(Object theProxy)**
  Returns the name of the .NET side bound to the specified proxy object. If the specified object is not a proxy, a DotNetSideException will be thrown.

As an example, assume that communications with the default .NET side is defined in the file dotNet1.properties, and communications with a second .NET side is defined in the file dotNet2.properties. The following example code configures the default .NET side, then creates a second .NET-side, then creates instances of a class C on each of the .NET-sides, then finally calls static class methods of C on each .NET-side:

```
DotNetSide.init("dotNet1.properties");      // defines "default"
DotNetSide.addServer("otherNSide", "dotNet2.properties");
C c1 = new C();  // created on the default .NET-side
DotNetSide.setServer( "otherNSide" );       // otherJSide is now active
C c2 = new C();  // created on otherNSide
c1.instanceMethod();    // sent to default .NET-side
c2.instanceMethod();    // sent to otherNSide
C.staticMethod();       // sent to otherNSide (currently active)
DotNetSide.resetDefaultServer();     // default is now active
C.staticMethod();       // sent to default .NET-side
```

Note that all the parameters in the above DotNetSide API calls can be variables, and that the active Java-side can therefore be determined programmatically.

Note on generating proxies on multiple .NET-sides: It is only possible to have the jnbproxy tool communicate with one .NET-side at a time. If all the .NET-sides offer the same set of classes, a single proxy jar file can be used to communicate with all .NET-sides if the DotNetSides API is used as above. If different .NET-sides offer different classes, one may create proxy jar files for each set using a separate run of the jnbproxy tool. The user is still responsible for indicating which .NET-side is to be used through the DotNetSides API. Also, if multiple proxy jar files have been generated, the same

class may be offered by multiple jar files. (In particular, System.Object and System.Type will be offered by each generated proxy jar file.)

Discovering information about .NET sides from the Java side

A Java-side client program can obtain information about the .NET sides with which it is communicating.

The method

static bool com.jnbridge.jnbcore.DotNetSide.isAlive(string *name*);

will return true if the .NET side with the supplied name is currently running, and false otherwise (for example, if it failed, or has been halted). If an invalid .NET side name has been supplied, a DotNetSideException will be thrown.

For convenience, the method

static bool com.jnbridge.jnbcore.DotNetSide.isAlive();

will return the status of the default Java side (which can also be queried with the first version of isAlive(), using the name "default".

The method

static String com.jnbridge.jnbcore.DotNetSide.CurrentServer()

returns the name of the currently active .NET side with which the Java-side client is communicating.

The method

static String com.jnbridge.jnbcore.DotNetSide.boundServer(Object *theProxy*)

returns the name of the .NET side bound to the specified proxy object. If *theProxy* is not a proxy object, a DotNetSideException will be thrown.

Discovering information about licensing

In many situations, it would be useful for an application to query the status of its JNBridgePro license. For example, an application could detect whether its JNBridgePro license is an evaluation license, and, if there are only a few days left, remind the user to purchase a valid license. To support this capability, JNBridgePro provides an API in the abstract class com.jnbridge.jnbcore.LicenseQuery, which contains the following methods:

- **static bool isValidLicense()** returns true if there is a valid license, throws a com.jnbridge.jnbcode.DotNetSideException otherwise.
- **static bool isTimeLimitedLicense()** if the license is valid, returns true if the license is time-limited (for example, an evaluation license), false otherwise. If the license is not valid, a com.jnbridge.jnbcode.DotNetSideException is thrown.
- **static int timeLeft()** if the license is valid and time-limited, returns the number of days left. If the license is invalid, a com.jnbridge.jnbcode.DotNetSideException is thrown, and, if valid but not time-limited, the result is undefined.
- **static string getInfo()** returns an descriptive string providing details about the license, useful for debugging. If the license is invalid, a com.jnbridge.jnbcode.DotNetSideException is thrown.

Note: The actual license query API is on the .NET side. When called from the Java side, the API is accessed through proxies, using JNBridgePro. This means that, if the license is invalid, JNBridgePro

will fail, which is why an exception is thrown. Note that it is also possible that the same exception could be thrown because JNBridgePro is incorrectly configured, or because the .NET side has not been started.

Mapping .NET entities to Java

This section describes the ways in which .NET classes and their members are mapped to Java proxies. The information will explain the structure of the proxies and how they can be used to access .NET entities from Java.

Directly-mapped classes

.NET primitive classes are directly mapped to Java classes as follows:

.NET	Java	Comments
int (System.Int32)	int	
short (System.Int16)	short	
long (System.Int64)	long	
float (System.Single)	float	
double (System.Double)	double	
char (System.Char)	char	
sbyte (System.Sbyte)	byte	The Java byte type is signed, as is the .NET sbyte.
bool (System.Boolean)	boolean	
byte (System.Byte)	short	There is no Java unsigned byte, so it's mapped to short to cover the range.
ushort (System.UInt16)	int	There is no Java unsigned short, so it's mapped to int to cover the range.
uint (System.UInt32)	long	There is no Java unsigned int, so it's mapped to long to cover the range.
ulong (System.UInt64)	float	There is no Java unsigned long, so it's mapped to float to cover the range.

In addition, the .NET class string (System.String) is directly mapped to the Java class java.lang.String.

.NET arrays of any rank or dimension, whose base elements are primitives or strings, are also directly mapped to Java arrays, for efficiency reasons. Thus, a .NET method that returns a value of, say string[][] will be represented in the Java proxy by a method that returns a Java array of type String[][]. Also note that .NET rectangular multidimensional arrays (for example, string[,]) are automatically mapped to equivalently-sized Java jagged arrays (in this example, String[][]).

To pass or return a string or array where a proxy method, field, or property expects a System.Object, you must wrap it in a DotNetString or DotNetArray class. See *Passing strings or arrays in place of objects*, below.

Enums

.NET enum classes and values are mapped directly to Java enum classes and values. When a .NET enum class is proxied, and the proxy generation tool's Java options are set so that Java 5.0-targeted

proxies are generated (see *Generating Java SE-5.0 targeted proxies*, above), the proxy assembly will contain a Java enum of the same name and containing the same set of enum values. If a .NET method returns an enum, the equivalent proxy method will return the corresponding Java enum, and if the method expects an enum as a parameter, the proxy method will expect the corresponding Java enum as a parameter. Similarly for setting and getting fields: if a field's type is a .NET enum, the proxy's corresponding field getter and setter will expect or return the corresponding Java enum.

To pass or return an enum where a proxy method or field expects a System.Object, you must wrap it in a DotNetEnum class. See *Passing strings, enums, or arrays in place of objects*, below.

Arrays

Arrays passed from the .NET to the Java side are automatically converted from .NET arrays to Java arrays. Similarly, Java arrays passed to .NET are converted from Java arrays to .NET arrays. This is done for reasons of efficiency: if a .NET method only returned an array reference to the Java side, rather than the entire array, then every access of an individual array element through a subscripting operation would involve a JNBridgePro round trip from the Java side to the .NET side and back. This would involve unacceptable overhead. By converting the entire array from .NET to Java, all element accesses are local, and much more efficient.

Arrays of any rank or dimension, whose base elements are primitives or strings are also directly mapped to .NET arrays, for efficiency reasons. Thus, a Java method that returns a value of, say, String[][] will be represented in the .NET proxy by a method that returns a .NET array of type string[][], where string is the .NET class string.

Class-name mapping algorithm

A Java proxy class has the same name as the corresponding .NET classes, and each proxy resides in a namespace whose name is identical to the package the underlying class resides in. For example, the .NET class System.Collections.Hashtable (class name Hashtable, namespace name System.Collections) is mapped to the Java proxy System.Collections.Hashtable (class name Hashtable, package name System.Collections).

Note that some Java development environments may complain when Java proxies in System.* namespaces are used, since they may erroneously detect a conflict with the System (java.lang.System) class. This is legal in Java and there is no conflict. One can usually get around this problem by explicitly importing the proxy (for example, import System.Type) and then referring to the class by its simple name (e.g., Type) in the source code.

If the .NET proxy corresponding to a Java class has not been generated (for example, if the /ns option has been chosen in jnbproxy, or if a Java call returns an object of an anonymous or dynamically generated class for which a proxy could not be generated in advance), a proxy for the object is generated dynamically and on the fly. This new proxy can be cast to any superclass it extends, or to any interface that it implements, and any method supported by the superclass or interface to which it has been cast can be called. This provides the complete class cast capability offered by both .NET and Java; with the previous proxy substitution scheme, certain class casts were not possible.

Note: Java-side proxies of .NET classes in the System namespace can cause errors and warnings when used in some Java development environments, because the namespace name conflicts with the class java.lang.System. To avoid these problems, simply import the full name of the class; for example:

import System.Object;

Following that, one can simply refer to System.Object as Object. In such a case, the Java System class must be explicitly referred to as java.lang.System, and the Java Object class must be explicitly referred to as java.lang.Object.

Proxy-mapped classes

All .NET classes, other than the directly mapped classes listed above, are mapped to generated proxy classes using the class-name mapping algorithm.

The .NET value null is directly mapped to the Java value null.

Access attributes of .NET classes are mapped directly to .NET access attributes, although the mapped attributes may not have identical meaning. Public .NET classes are mapped to public Java classes. Non-public classes are not mapped to Java proxies. Sealed .NET classes are mapped to final Java proxy classes. Static .NET classes are mapped to final Java proxy classes without a public constructor.

The superclass of a .NET class is mapped to the superclass's Java proxy class using the class-name mapping algorithm, described above.

Similarly, all of a class's interfaces are mapped to proxy interfaces of the same name. If a proxy interface has not been generated, the interface is not mapped.

For each field F in a .NET class, getter and setter methods Get_F() and Set_F() are placed in the generated proxy. Get_F() takes no arguments and returns a value of the corresponding field's type (as transformed by the class mapping algorithm). Set_F() is a void method that takes one argument whose type is the type of the corresponding field (as transformed by the class mapping algorithm). Public and protected attributes of the field are mapped to public and protected attributes of the corresponding getter/setter methods in the proxy. Private and default-access fields are not mapped to the proxy. .NET fields that are static are mapped to static getter/setter methods in the Java proxy. If a field in the .NET class is sealed, final, or readonly, only a getter method is generated in the Java proxy; no setter method is generated. If a field in the .NET class is literal (that is, is a compile-time constant), no getter or setter method is generated in the proxy, but instead a constant field of the same name is generated in the proxy, and it is initialized value of the underlying .NET field.

For each method and constructor in a .NET class, a method or constructor of the same name is placed in the generated proxy. Public, protected, and static access attributes of the method or constructor are mapped to public, protected, and static attributes of the corresponding method or constructor of the proxy. Private, internal, and default-access methods and constructors are not mapped. Methods that are static and/or abstract in the Java class are made static in the .NET proxy. Classes of method return values are mapped according to the class name mapping algorithm. *If a method or constructor has a parameter or a class for which a proxy has not been generated, that method or constructor will not be included in the generated proxy.*

Methods overriding final java.lang.Object methods: java.lang.Object has six methods that are **final**: getClass(), notify(), notifyAll(), wait(), wait(long), and wait(long, int). That means that no other Java class (all of which inherit from java.lang.Object) can declare methods with those signatures. Either the Java compiler will detect the error, or the Java runtime verifier will (for classes that are not generated using the compiler). If a .NET class is proxied and it has a method with one of these signatures, the proxied method will be renamed so that the suffix *__JNBridgeProxy* is appended. Thus, if the .NET method has a method wait(long, int), the proxied method on the Java side will be

wait__JNBridgeProxy(long, int). Renaming those methods in this way will allow them to pass the Java verifier.

Java SE 5.0and later only: If a method or constructor in the underlying .NET class has a variable number of arguments(that is, if it is of the form f(params int[] x)), then the corresponding proxy method will also have a variable number of arguments (that is, will be of the form f(int… x)). Note that this mapping is only performed if the "Generate Java SE 5.0-targeted proxies" option is chosen before generating the proxies.

As a convenience, calls to the method toString() in the generated Java proxies (a method supported by all Java objects including the proxies) are automatically passed through to the .NET classes' ToString() method, thereby returning the underlying .NET object's string representation, rather than the string representation of the proxy object. As before, ToString() can still be called directly through the proxies. Similarly, calls to the methods equals() and hashCode() in the generated Java proxies are automatically passed through to the .NET classes' Equals() and GetHashCode() methods. See the section "Proxy implementation of equality and hashcode methods," below.

For each property P in a .NET class, getter method Get_P() is placed in the generated proxy if P has a **get** method, and similarly Set_P() is placed in the generated proxy if P has a **set** method. Get_P() takes no arguments and returns a value of the corresponding property's type (as transformed by the class mapping algorithm). Set_P() is a void method that takes one argument whose type is the type of the corresponding property (as transformed by the class mapping algorithm). Public and protected attributes of the property are mapped to public and protected attributes of the corresponding getter/setter methods in the proxy. Private, internal, and default-access properties are not mapped to the proxy. .NET properties that are static are mapped to static getter/setter methods in the Java proxy. If a property in the .NET class is sealed or final, the getter and/or setter methods in the proxy will be final.

If a .NET property P is an indexer, the generated getter method Get_P()'s parameters will be the parameters of the indexer, and the generated setter method Set_P()'s parameters will be the value to be assigned to the property, followed by the parameters of the indexer. (***Note***: this order is different from that in v2.x; it was changed in order to accommodate indexers with a variable number of arguments.) For example, the .NET class System.Collections.Hashtable has an indexer property Item. A Java proxy for System.Collections.Hashtable would have methods

public System.Object Get_Item(System.Object index);
public void Set_Item(System.Object value, System.Object index);

Java SE 5.0 and later only: If an indexer property in the underlying .NET class has a variable number of arguments(that is, if it is of the form this[params int[] x]), then the corresponding proxy indexer method will also have a variable number of arguments (that is, will be of the form Get_P(int… x) or Set_P(OutputType v, params int… X). Note that this mapping is only performed if the "Generate Java SE 5.0-targeted proxies" option is chosen before generating the proxies.

The methods, constructors, and property getters/setters of the generated proxies will have metadata assigning their parameters the same names as the parameters of the methods, constructors, and property getters/setters of the underlying .NET classes. These names will show up when the proxies are used in Java IDEs such as Eclipse.

Due to the fact that multiple .NET primitives may be mapped to the same Java types (for example, int and ushort), it is possible that the mapping algorithm will generate multiple methods with the same signature. For example, a .NET class with two methods f(int) and f(ushort) will map to a Java proxy whose methods both map to f(int). Since multiple methods with the same signature are not legal in

Java, the proxy generator will drop all but one of such multiple methods. The JNBProxy proxy generator will notify the user of which methods have been dropped.

Note: Any .NET class member returning a pointer type or expecting one as a parameter will not be included in a Java-side proxy of that class.

Explicit interface implementations

If the underlying .NET class has a method that is an *explicit interface implementation*, it will be proxied in the same way as any other method, and the proxy method's name will be the simple name of the implemented method. For example, if there is a C# interface IExample:

```
public interface IExample
{
    public void IExample f();
}
```

and a class C that explicitly implements it:

```
public class C : IExample
{
    void IExample.f() { }
}
```

the method IExample.f() in C can be accessed in the following ways:

```
C c = new C();
IExample ie = c;
// either of the following will work:
c.f();
ie.f();
```

The one exception is if the class contains another method f() that is not an explicit implementation. For example:

```
public class C : IExample
{
    void IExample.f(){ }

    public string f() { return "hello"; }
}
```

In that case, the ordinary method f() (the one that's not an explicit interface implementation) will be proxied, and the explicit interface implementation will not.

The proxy generation tool will report that it did not proxy the method in the proxy tool's output at the end of the proxy generation run:

```
The following class members could not be generated when mapped to Java
proxy methods because, after mapping, their signatures were identical to
proxy methods already generated:
…
Method: C.IExample.f()
…
```

That means that a call to c.f() in the above example will return the string "hello", and a call to ie.f() will throw an AbstractMethodError, since the actual method has not been implemented for the interface. However, the missing explicit interface implementation can still be called from the Java side through .NET reflection:

```
// System.Type and System.Object are always proxied; make sure that BindingFlags is proxied, too
import System.Type;
import System.Object;
import System.Reflection.BindingFlags;
…
Type interfaceType = IExampleHelper.GetDotNetClass();
interfaceType.InvokeMember("f", BindingFlags.InvokeMethod, null, c, new Object[0]);
```

The above also applies to explicit interface implementations that are properties or events.

Abstract classes

Previous to JNBridgePro v9.0, it was not permitted to have a Java class that inherited from a proxy of a .NET abstract class: if such a Java class was instantiated, JNBridgePro would attempt to instantiate the underlying .NET abstract class, and an MemberAccessException would be thrown. Beginning with JNBridgePro v9.0, this no longer occurs, and it is permitted to write a Java class that inherits from a proxied .NET abstract class.

Interfaces

.NET interfaces are mapped into Java interfaces. Interfaces are mapped in the same way as classes, with two exceptions. First, the class name mapping algorithm is modified so that if no resulting interface is found as a result of following the superinterface chain, the interface is not mapped. This is done because interfaces are not rooted in `System.Object` and so a result is not guaranteed. If such a result occurs, it is recommended that you regenerate the proxies to include all required interfaces.

Second, the method GetDotNetClass() (implementing the .NET-side class literal – see "Class literals," below) clearly cannot be implemented in an interface, so for each interface I, a corresponding abstract class IHelper will be generated, containing exactly one method, the static method GetDotNetClass(), which, when called, will return the proxy of the .NET type object for the corresponding interface I.

Generics

When generating a Java-side proxy from a .NET generic class, the proxy that is generated is based on a non-generic *flattening* of the original class. This flattened class is obtained by removing the generic's type parameters, and by replacing all instances of the type variables with the variables' upper bounds. If any members of the class have types or parameters which are themselves generic types, they are also replaced with flattened types. For example, if a proxy is generated for the following .NET generic class:

```
public class GenericClass<T, U> where T : System.Int32
{
        public U aField;
        public List<Integer> anotherField;

        public static U aStaticField;

        public T aMethod(Collection<Float> c) { … };
        public GenericClass(int i){…};
}
```

the flattened Java proxy class will have the following signature.

```
public class GenericClass__2
{
        public System.Object Get_aField();          // field getter
        public void Set_aField(System.Object);      // field setter

        public List__1 Get_anotherField();                  // field getter
        public void Set_anotherField(List__1);      // field setter

        public System.Object Get_aStaticField(System.Type, System.Type);        // field getter
        public void Set_aStaticField(System.Type, System.Type, System.Object); // field setter

        public System.Int32 aMethod(Collection__1);

        public GenericClass__2(System.Type, System.Type, int);
}
```

Most of the rules for creating Java proxies from .NET generics are the same as those for generating proxies from conventional non-generic classes (see the section "Proxy-mapped classes," above), but there are some significant differences:

- The names of Java proxies for .NET generic classes have the following form: *name__nTypeParams*, where *name* is the base name of the generic class, and *nTypeParams* is the number of type parameters in the generic class, and they are separated by two underscores ('__'). Thus, the .NET class GenericClass<T,U> (also known as GenericClass`2), maps to a Java proxy GenericClass__2. (The reason for this is that the internal .NET name of a generic class uses the backquote `, and the backquote is not a legal character in a Java identifier.)
- All constructors for a generic class take a sequence of System.Type objects (or rather, their proxies) corresponding to the generic's type arguments as parameters before any of the original constructors' parameters. Thus, if GenericClass<T,U> has a constructor GenericClass(int i), the proxy will have a corresponding constructor GenericClass__2(System.Type T, System.Type U, int i). The reason for this is that, unlike Java generics, whose type parameter information is removed at compile time, type parameters for .NET generics exist at run time. In this case, we must specify to the .NET side the bindings we want assigned to the generic invocation. For example, if A and B are .NET classes with Java proxies, we can create an instance of GenericClass<A,B> at run time from the Java side by executing **new** GenericClass__2(A.GetDotNetClass(), B.GetDotNetClass(), *i*), where *i* is some integer. Note that one consequence of this transformation is that Java proxies of .NET generic classes will not have a default constructor.
- If a .NET generic class has static members, then each corresponding static member of the Java proxy will take a sequence of System.Type objects (or rather, their proxies) corresponding to the

generic's type arguments as parameters before any of the other parameters in proxy's corresponding member. This is because static members of .NET generics must be referenced as members of a particular invocation of the generic rather than the generic itself. For example, to access GenericClass<T,U>'s aStaticField, one must assign values to T and U (for example, GenericClass<A, B>.aStaticField, where A and B are classes). To do the same thing through the Java proxy, one must execute GenericClass__2.Get_aStaticField(A.GetDotNetClass(), B.GetDotNetClass()). *Note that the generic type objects must be supplied even if the method call is made in the context of a proxy object. For example, if the object x is of type GenericClass<A, B>, a call to Get_aStaticField() on its Java-side proxy must be of the form x.Get_aStaticField(A.GetDotNetClass(), B.GetDotNetClass), even though the generic type parameters have already been bound in the context of x. x's generic type parameter bindings are ignored; the only ones that matter are the ones supplied with the call.*

Similarly, if a proxy is generated for an otherwise non-generic class with a generic method, the proxy is generated using the method in its raw form. For example, for a generic method

```
public V aMethod<V>(V v) where V : System.Int32 { … }
```

the corresponding proxy method has the signature

```
public System.Int32 aMethod__1(System.Type V, System.Int32 v) ;
```

where V corresponds to the type parameter of the underlying generic method, and __1 denotes that the method has one generic parameter. In addition, if the generic method is a static member of a generic .NET class, its will have the declaring class's type parameters at the beginning of its parameter list. For example, if the above aMethod() method were a static member of the class GenericClass<T,U>, the corresponding proxy method would have the signature

```
public static System.Int32 aMethod__1(Systen.Type T, System.Type U, System.Type V,
System.Int32 v) ;
```

It is important to note that only the type parameters' upper bounds are mapped to the Java proxies; all other constraints, including interface, default constructor, and class/struct constraints are ignored. If a type is passed that violates a constraint, a System.ArgumentException will be thrown.

Note: The reason that Java proxies of .NET generic classes are not themselves generic but rather are flattened classes with additional parameters supplying type information is that Java and .NET generics are based on very different models. Java generics are based on a compile-time technique known as *erasure*, where all the generic information is removed during compilation, and a least general non-generic class that covers all the allowable cases is substituted. With Java generics, type parameter information does not exist at run time. Also, all invocations of Java generic classes are really the same class, even if the type parameters had different values .NET generics, on the other hand, are based on a very different model where invocations of generic classes with different type parameters are different classes, and where type parameter information exists and is accessible at run time. Thus, .NET generics could not be mapped directly into Java generics, and it is necessary to add the additional type arguments to the proxy's constructors and static members so that the type information can be supplied at runtime.

Cross-platform overrides

When a .NET class is proxied to the Java side, one can create Java subclasses that inherit from the proxied .NET class. It is possible to pass objects of that Java subclass back to the .NET side, since JNBridgePro considers them to be of the proxied type. On the .NET side, any calls to methods that were overridden in the Java subclass will be redirected to the Java-side code in the Java subclass.

(Note that this behavior was new in JNBridgePro v9.0. In previous JNBridgePro versions, calls to overridden methods were not redirected back to the Java side.)

Only methods that are considered overridable in .NET (that is, instance methods marked as virtual that are not marked as sealed) can be overridden in Java-side subclasses of proxy classes.

To make use of this override functionality, any Java class containing methods overriding proxied methods must contain a call to com.jnbridge.jnbcore.Overrides.registerOverrides() inside its static initializer. If there is no static initializer in the Java subclass, you should add one. Ideally, the call to Overrides.registerOverrides() should be the first call in the static initializer.

Note that, if you call Overrides.registerOverrides() in a class's static initializer, then any instance of this class will by default not be garbage collected before the end of the program. The reason for this is to avoid situations where the .NET object is garbage collected before an overriden method is called on the Java side. If you are sure that your .NET object is no longer needed, and that it is safe to garbage collect it, you can call com.jnbridge.jnbcore.Overrides.unregisterOverrideObject(Object), passing as a parameter the object that can now be garbage collected. Note that it is always safe to call unregisterOverrideObject() more than once on an object, or when the object is not an override object. Calling unregisterOverrideObject() with a null argument will result in an exception.

As an example of using overrides, consider the following example:

```
// C# code
public class DotNetBaseClass
{
        public virtual void method1()
        {
                Console.WriteLine("DotNetBaseClass.method1");
        }

        public virtual void method2()
        {
                Console.WriteLine ("DotNetBaseClass.method2");
        }
}

public class DotNetFrameworkCode
{
        private static DotNetBaseClass myClass = null;

        public static void registerObject(DotNetBaseClass dnbc)
        {
                myClass = dnbc;
        }

        public static void runFramework()
        {
                myClass.method1();
                myClass.method2();
        }
}

// Java code
public class JavaSubClass extends DotNetBaseClass
{
        static
        {
                com.jnbridge.jnbcore.Overrides.registerOverrides();
        }

        public void method1()
        {
                System.out.println("DotNetSubClass.method1");
        }

        public void method2()
        {
                System.out.println("DotNetSubClass.method2");
                super.method2();          // call DotNetBaseClass.method2()
        }
}
```

When we run the following code from the .NET side:

```
DotNetBaseClass dnbc = new JavaSubClass();

DotNetFrameworkCode.registerObject(dnbc);

DotNetFrameworkCode.runFramework();
```

we will see the following output:

```
JavaSubClass.method1
JavaSubClass.method2
DotNetBaseClass.method2
```

Callbacks (events and delegates)

.NET supports callbacks through the use of *delegates* and *events*. (A discussion of delegates and events is beyond the scope of this manual. Please see the .NET documentation for more information.) JNBridgePro allows the developer to register Java classes as delegates and event handlers in order to get callback functionality in Java-to-.NET projects.

Delegates

A *delegate* is a special kind of .NET type used to support callbacks. In order to register a Java class as a .NET delegate, one must first generate a Java-side proxy for the .NET-side delegate type. The generated proxy will be an interface with a single method called Invoke(), whose signature is identical to the delegate's signature. For example, if there is a .NET delegate

```
// .NET-side code
public delegate int SampleDelegate(int x, float y);
```

and a .NET class using it:

```
// .NET-side code
public class C
{
        public SampleDelegate sampleDelegate;

        public void fire()
        {
          int z = sampleDelegate(1, 2.0F);
        }
}
```

the proxy for SampleDelegate will be a Java interface

```
// Java-side code
public interface SampleDelegate
{
        int Invoke(int x, float y);
}
```

To create and register a Java-side delegate, make sure that you've generated proxies for the delegate type and the classes in which the delegate is used (in this case, SampleDelegate and C, respectively). Then, create a class that implements the delegate's Java-side proxy interface:

```
// Java-side code
public class JavaSampleDelegate implements SampleDelegate
{
        // add any constructors here

        public int Invoke(int x, float y)
        {
         System.out.println("Invoke called: x = " + x + ", y = " + y);
         return x+1;
        }
}
```

Next, write code that creates an instance of the Java-side delegate class and assigns it to the .NET-side delegate field:

```
// Java-side code
SampleDelegate s = new JavaSampleDelegate();
…
// let c be a proxy of an instance of the class C
c.set_sampleDelegate(s);
```

Now, when c.fire() is called, the Invoke() method of s, the instance of JavaSampleDelegate, is called.

Note: Although .NET supports the use of both static and instance methods as delegates, with JNBridgePro your callback method must be an instance method named Invoke(). Also note that a called Java-side delegate can return a value or throw an exception back to its .NET-side caller.

Java callbacks have the following restrictions:

- All parameters of the Invoke methods in the implemented delegate interfaces must be of types recognized by the .NET side. These include primitives (int, long, short, byte, char, boolean, float, and double), strings, .NET objects for which Java proxies have been generated, and arrays of any of the above.
- Values returned by callback methods must be of types recognized by the .NET side. These include primitives (int, long, short, byte, char, boolean, float, and double), strings, .NET objects for which Java proxies have been generated, and arrays of any of the above.
- Exceptions thrown by callback methods must be of types for which a Java proxy class has been generated. If any other exception is thrown, the result is undefined.
- Callback objects may not be returned to the Java side as a result from a call to a .NET method, or retrieved by the Java side from a .NET field. This restriction will be lifted in a future version.
- Callback objects may not be passed to the .NET side where an object of type System.Object or any other type of .NET object is expected, since they do not inherit from System.Object. They may only be passed where an object that implements the same listener interface is expected. This restriction will be lifted in a future version.

Events

An event is a special kind of delegate with some additional capabilities. An event may be a field or a property of a class that holds a delegate. For example:

```
// .NET-side code
public class C
{
```

```
        public event EventHandler myEvent;

        public event EventHandler MyEventProperty
        {
         add { … }
         remove { … }
        }

        public void fire()
        {
         myEvent(this, new EventArgs());// or MyEventProperty(this, new EventArgs());
        }
}
```

In the example above, EventHandler is a delegate type, and any delegate may be used in that place. As with delegates, Java classes can be registered as event handlers using a similar mechanism.

First, make sure that Java-side proxies have been generated for the event handler delegate type and the classes in which the events are declared (in this case, EventHandler and C, respectively).

Next, on the Java side, create an event handler class that implements the EventHandler interface:

```
// Java-side code
public class JavaEventHandler implements EventHandler
{
        // constructors can go here

        public void Invoke(System.Object sender, EventArgs e)
        {
         // event handling code goes here
        }
}
```

Finally, instantiate the Java event handler and add it to the set of event handlers:

```
// Java-side code
EventHandler j = new JavaEventHandler();
…
// let c be a proxy of an instance of C
c.add_myEvent(j);         // or c.add_MyEventProperty(j);
```

Then, calling c.fire() will cause the Invoke() method of j, the JavaEventHandler, to be fired along with any other event handler registered with myEvent or MyEventProperty.

It is also possible to remove a registered event handler by using a remove method. For example, one can call c.remove_myEvent(j) or c.remove_MyEventProperty(j) to remove j from the set of registered event handlers for myEvent or MyEventProperty.

Note: Although .NET supports the use of both static and instance methods as event handlers, with JNBridgePro your callback method must be an instance method named Invoke(). Also note when an event has multiple registered event handlers, all the returned values and thrown exceptions from called event handlers are discarded except for the last registered event handler that is called. Since the order in which events are registered is often unknown, it is recommended that one not try to return values or throw exceptions from event handlers.

Asynchronous callbacks

When callbacks involve Java or .NET windows and forms, it is possible that use of callbacks can lead to deadlock between the Java and .NET sides. One way to avoid the possibility of deadlocks is to use an *asynchronous callback*. Asynchronous callbacks are exactly like regular callbacks except that the .NET thread invoking the callback does not suspend until the callback completes, and does not wait to see if a value is returned or an exception is thrown back. This means that Java calls to .NET methods leading to callbacks never suspend and deadlock is not a possibility.

Asynchronous callbacks cannot return values or throw exceptions back to the .NET side. Any values returned by callback methods are ignored, and any exceptions thrown by callback exceptions are also ignored. Since callbacks can themselves call .NET methods, this can be used to get the same effect as returning values to the .NET side directly.

To designate an asynchronous callback, a Java-side callback class must implement the com.jnbridge.jnbcore.AsyncCallback marker interface, which contains no members. Any class that implements AsyncCallback is an asynchronous callback object; if it is a callback object and does not implement AsyncCallback, then it is not an asynchronous callback object.

Managing callback resources

Each callback object that is registered with the .NET side consumes resources, including a listener thread. Even when a callback object is unregistered and the object is garbage-collected, these resources still persist, unless explicit action is taken. If an application only uses a few callbacks, this should not matter, but if the application creates many callbacks, the application may run out of resources unless they are explicitly managed.

To manage callback-related resources, we have provided an API:
static void com.jnbridge.jnbcore.Callbacks.releaseCallback(Object callbackObject).

After the callback object has been unregistered with the .NET side and will no longer be used as a callback, the user can call Callbacks.releaseCallback() with the callback object as its argument. This will release the resources associated with the callback, and will avoid resource-related problems when many callbacks are created. The callback object supplied as a parameter must implement a proxied delegate or event interface. Note that releaseCallback() really expects an object that implements a special marker interface that is implemented by every proxied delegate or event interface; if an object that does not implement a delegate or event interface is supplied as a parameter, a compilation error will result.

The callback object cannot be used as a callback after releaseCallback() has been called with it as a parameter. If the object is used as a callback after that point, an exception will be thrown. It is still possible to use the object in other ways (other than as a callback) after releaseCallback() is called.

Nested classes and interfaces

JNBridgePro supports access of .NET nested classes and interfaces from Java. If the user adds a .NET class to the JNBProxy environment, and the class has nested classes or interfaces, the nested classes are also added. This process is done recursively, so if the nested classes themselves have nested classes, those further nested classes are themselves added.

It is also possible to explicitly add a nested class. For example, if the .NET class A encloses a class B, one can explicitly add A.B to the set of proxies by requesting that JNBProxy add the class A+B. ('+' is the internal .NET delimiter between enclosing and nested class names.) If a nested class is

added, its enclosing class or interface is also automatically added, as well as any classes or interfaces that nest within it. Again, the process is recursive, and proceeds until a fixpoint is reached.

.NET nested classes do not work in exactly the same way as Java nested classes, and so they do not map directly. All .NET nested classes corresponding to Java static nested classes, so for any nested .NET class for which a Java proxy is generated, the Java proxy will be a static nested class.

Structs and enums

A Java proxy can be created for a .NET struct (value class). The proxy should look and behave exactly the same as if the underlying .NET class were a value class, and the Java programmer should not notice any difference.

Java proxies can be created for .NET enumerated types (enums). Proxy for an enumerated type will have all the methods defined for the enumerated type, plus a literal field for each enumerated value in the type. For example, if a proxy is created for the .NET class System.DayOfWeek, the proxy class will have static fields DayOfWeek.Sunday, DayOfWeek.Monday, DayOfWeek.Tuesday, etc., all of which are of type System.DayOfWeek.

Reference and out parameters

.NET languages can contain methods with ref and out parameters; this is a mechanism to return data by changing the value of one or more arguments used in the call. Since Java does not provide an equivalent mechanism, we have provided a different mechanism for passing back the values that parameters should have upon return.

If a .NET method takes one or more ref or out parameters, the Java proxy of the method will have the same signature, except that it will now return a value of type java.util.Hashtable, regardless of the return type of the original .NET method. For example, if we have a .NET method

```
public int f(int x, out int y)
{
    int z = x + y;
    y = 10;
    return z;
}
```

the proxied method will have the signature

java.util.Hashtable f(int x, int y)

When you call a proxied method whose underlying method contains ref or out parameters, the proxy returns a hash table containing the returned result (if there is one), and the final values for all the ref or out arguments. The return value is accessed through the key "result" (a string), and the changed arguments are accessed through a string denoting the zero-based position of the argument (for example, "0", "1", and so on). For example, the above proxy can be called as follows:

Hashtable h = f(1, 2);

The returned hash table will have the following entries:

- key = "result", value = java.lang.Integer(3)
- key = "1", value = java.lang.Integer(10)

Note the following:

- When a primitive value is returned through the hash table (for example, the int values above), the value in the hash table is wrapped in the appropriate wrapper class. In all other cases, the appropriate proxy or mapped object is returned.
- If an argument is neither ref nor out, its final value is not returned in the hash table. For example, the first argument above, which would correspond to key "0" is not ref or out, so its value will not change, and there is no need to return it in the hash table.
- In .NET programs, an argument corresponding to an out parameter must usually be a variable that can take the changed value, and attempting to call with any other parameter will cause a compilation error. When calling the corresponding proxy from Java, this restriction does not apply, and one can supply any valid expression as the argument, as in the example above.
- If the underlying method returns a null value, or the underlying method has a void return type, the "result" entry in the returned hash table will have the singleton value of type com.jnbridge.jnbcore.NullValue. This is because one cannot use null as the value in a Java hash table (although it is allowed in .NET hash tables).
- It is the responsibility of the user to know the return type of the called method that contains ref/out parameters, as well as which parameters are ref/out and what their types are, and to perform the appropriate type casts and unwrapping operations as required.

Pointer types

If a .NET class contains a member that returns a pointer type or expects one as a parameter, the Java-side proxy of that class will not contain that member.

Class literals

If a .NET class C has a Java proxy, executing C.class on the Java proxy will return the Java-side Class object for the proxy. To access the .NET-side class object for C (the equivalent of executing typeof(C) on the .NET side) from Java, execute C.GetDotNetClass() on the Java proxy, and a Java proxy object of class System.Type, representing the .NET class C, will be returned.

For an interface I, executing IHelper.GetDotNetClass() will return the proxy for I's .NET type.

Exceptions

Exceptions can be thrown from the .NET side to the Java side. To throw an exception across the .NET-Java boundary, make sure that proxy classes have been generated for each exception that might be thrown. Then, when a .NET exception is thrown, the Java code can catch the corresponding proxy exception object.

Since .NET methods do not indicate which exceptions they throw (unlike Java methods), exceptions thrown by methods cannot appear in the set of supporting classes for a given .NET class. Also, since we do not know what exceptions a .NET method will throw, the proxy's method will not specify any thrown methods. Consequently, all proxy exceptions thrown from .NET to Java must be *unchecked* exceptions (that is, need not appear in the **throws** clause of a method definition if they are not handled in the method body). In order to be unchecked exceptions, all proxies for .NET exceptions inherit from java.lang.RuntimeException; all RuntimeExceptions are unchecked.

If a proxy was not previously generated for a thrown exception, the nearest superclass for which a proxy has been generated will be used, up to the proxy for System.Object, which is always generated.

To access the exception message or stack trace associated with the thrown Java exception, access the string values in the property getters Get_Message() and Get_StackTrace() associated with the corresponding .NET proxy object. Java proxies for .NET exception classes are subclasses of the java.lang.Throwable class, and themselves have getMessage() and printStackTrace() methods, but these are generated by Java-side code and contain Java-specific information.

Proxy implementation of equality and hashcode methods

JNBridgePro maps Java-side proxies' equals() and hashCode() methods to the corresponding .NET objects' Equals() and GetHashCode() methods. This allows them to behave as expected when added to Java hashtables and other collections.

Extension methods

.NET supports ***extension methods***, which are a way to add methods to an existing class without actually altering the class; the extension methods are really specially constructed methods in a different class. Extension methods are purely a compile-time construct; they do not exist as such at run time.

If a class has an extension method, it will not appear as a proxied method in the proxy of the class, but you can still access it through the original class. For example, Concat() is an extension method of the generic collection List<>. As an extension method, it will not appear in the Java proxy class List__1. However, Concat() is actually a static method in System.Linq.Enumerable, and can be accessed through a proxy of Enumerable. This means that you cannot write a proxy call myList.Concat(myOtherList), but you can write the equivalent proxy call Enumerable.Concat(myList, myOtherList).

Passing strings, enums or arrays instead of objects

For reasons of efficiency and convenience, generated JNBridgePro proxies translate returned .NET string objects into native Java Strings. Similarly, returned .NET array objects are translated into native Java arrays. Conversely, users pass proxies native Java Strings and arrays when the underlying .NET methods and fields expect .NET strings and arrays.

When a .NET method is declared to return an Object (that is, a System.Object), and the method actually returns a string, it is not possible to automatically convert the .NET string to a Java String because the Java String is not a subclass of System.Object. In such cases, the method returns an object of class System.DotNetString, which is a subclass of System.Object and which wraps the native Java String. To obtain the native Java string, call the DotNetString property stringValue().

For example, assuming the following .NET class:

```
public class c
{
  public string retString() { return "abc"; }
  public Object retObj() { return "def"; }
}
```

Java code would access a proxy for c as follows:

```
c o = new c();
String s = o.retString();   // s contains .NET string "abc"
```

```
System.DotNetString ns = (System.DotNetString) o.retObj();
String s2 = ns.stringValue();  // s2 contains Java String "def"
```

When a method declared to return a .NET Object returns an array, the proxy will return an object of class System.DotNetArray, a subclass of System.Object. The DotNetArray object wraps the native Java object, which can be retrieved by calling the method arrayValue(). arrayValue() returns a Java Object, which is the base class for all Java arrays. Therefore, assuming the following .NET class:

```
public class d
{
  public int[] retArray() { return new int[3] { 1, 2, 3 }; }
  public Object retObj() { return new int[3] { 4, 5, 6 }; }
}
```

Java code would access a proxy for d as follows:

```
d o2 = new d();
int[] a = o2.retArray();   // a contains Java array { 1, 2, 3 }
System.DotNetArray na = (System.DotNetArray) o2.retObj();
int[] a2 = (int[]) ja.arrayValue();  // a2 contains Java array { 4, 5, 6 }
```

The method arrayValue() may only return arrays whose elements are primitives or objects descended from System.Object. Therefore, for example, when a method declared to return Object returns an array of strings, the proxy will return a DotNetArray containing DotNetStrings, as follows:

```
public class e
{
  public string[] retArray() { return new string[] { "a", "b", "c" }; }
  public Object retObj() { return new string[] { "d", "e", "f" }; }
}
```

Java code would access a proxy for e as follows:

```
e o3 = new e();
String[] a = o3.retArray();   // a contains Java array { "a", "b", "c" }
System.DotNetArray na = (System.DotNetArray) o3.retObj();
System.DotNetString[] a2 = (System.DotNetString[]) na.arrayValue();
// a2[0].stringValue() == "d"
// a2[1].stringValue() == "e"
// a2[2].stringValue() == "f"
```

When accessing fields or properties declared to be of class System.Object, the rules are the same as for methods that are declared to return values of System.Object.

When a method expects a parameter of class System.Object, one cannot pass a Java string or array, but must wrap the string or array in a DotNetString or DotNetArray object. For example, if there is a .NET class f as follows:

```
public class f
{
  public void passString( string s ) { }
  public void passArray( int[] i ) { }
  public Object passObj( Object o) { }
}
```

.NET code would access a proxy for f as follows:

```
f o4 = new f();
String s = "a";
```

```
int[] i = {1, 2, 3};
o4.passString( s );
o4.passArray( i );
o4.passObj( new System.DotNetString( s ) );
o4.passObj( new System.DotNetArray( i ) );
```

Unlike the case of methods returning arrays or strings nested in arrays, it is not necessary to wrap the inner strings or arrays inside DotNetString or DotNetArray objects; the system will automatically take care of this. Thus, the following call is legal:

```
o4.passObj( new System.DotNetArray( { "a", "b", "c" } ) );
```

When assigning to fields or properties declared to be of class System.Object, the rules are the same as for passing strings or arrays in place of Object parameters.

Similarly, when a proxied field or method is declared to return a System.Object, and an enum is returned, it is returned wrapped inside a System.DotNetEnum object. The actual native Java enum can be retrieved by accessing the DotNetEnum object's enumValue() method. Likewise, when a proxied method has a declared parameter of type System.Object, or a field is of type System.Object, and an enum is passed to the method or assigned to the field, it is necessary to wrap the enum in a DotNetEnum object before passing it to the method or assigning it to the parameter.

For example, if there are .NET classes g and h as follows:

```
public enum g { x, y, z };

public class h
{
  public object passObj( object o) { }
}
```

The proxy generator would create a native Java enum class g and a proxy class h. Java code would access the proxy for h as follows:

```
h o5 = new h();
System.DotNetEnum o6 = (System.DotNetEnum) o5.passObj( new System.DotNetEnum( g.x ) );
g g1 = (g) o6.enumValue();        // g1 is now g.x
```

The DotNetArray, DotNetString, and DotNetEnum wrapper classes implement proxies of the same interfaces as the underlying .NET classes. For example, since .NET arrays implement the IList`1 interface, DotNetArray implements the proxy of the IList`1 interface. The purpose of this is to allow wrapper objects to be passed or returned where the proxy is declared to pass or return an object of the interface. Note that although the wrapper classes impement the proxied interfaces, the wrapper classes don't actually implement the interfaces' methods, and if an interface method is called on a wrapper object, an exception will be thrown.

Boxed primitive types

.NET offers the ability to transparently pass or return primitive values where System.Objects are expected. It does this through a mechanism called *boxing*. If a primitive needs to go where a reference object such as Object is expected, .NET automatically wraps the primitive in a *boxed value*, then automatically unwraps it when the value must go where a primitive is expected. The mechanism is similar to Java's wrapper classes such as java.lang.Integer, except that in .NET boxing and unboxing is done automatically and transparently.

JNBridgePro provides support for the boxing and unboxing of primitives. When a .NET method returns a primitive (say, an int) where a System.Object is expected, the Java proxy for that method will return an object of type System.BoxedInt. The primitive value can be accessed by executing the method intValue(), which returns the primitive integer.

Similarly, when a .NET method expects a parameter of type System.Object, and the developer wishes to pass the Java proxy method a primitive value, the primitive can be boxed. For example, to pass an integer 3 in to f(System.Object o), one can call f(new System.BoxedInt(3)).

All boxed value objects inherit from the abstract class System.BoxedValue, and all boxed value classes implement the instance method unboxToJava(), which is declared to return a value of type java.lang.Object, and which returns the equivalent Java wrapped object. For example, calling unboxToJava() on a System.BoxedInt will return a java.lang.Integer containing the same primitive integer.

The following boxed value classes are supported by JNBridgePro:

Primitive type	Boxed type	Accessor method
int (System.Int32)	System.BoxedInt	intValue()
short (System.Int16)	System.BoxedShort	shortValue()
long (System.Int64)	System.BoxedLong	longValue()
float (System.Single)	System.BoxedFloat	floatValue()
double (System.Double)	System.BoxedDouble	doubleValue()
char (System.Char)	System.BoxedChar	charValue()
sbyte (System.SByte)	System.BoxedSByte	sbyteValue() (returns byte)
bool (System.Boolean)	System.BoxedBoolean	booleanValue()
uint (System.UInt32)	System.BoxedUInt (parameters) System.BoxedLong (return values)	longValue()
ushort (System.UInt16)	System.BoxedUShort (parameters) System.BoxedInt (return values)	intValue()
ulong (System.UInt64)	System.BoxedULong (parameters) System.BoxedFloat (return values)	floatValue()
byte (System.Byte)	System.BoxedByte (parameters) System.BoxedShort (return values)	shortValue()

Note that, for uint, ushort, ulong, and (unsigned) byte, there are two boxing wrapper classes. The choice of which is used depends on whether it is being passed as a parameter of a proxy method, or is being returned from a proxy method.

Reference and value proxies

JNBridgePro supports the ability to designate proxies as "reference" or "value." A Java reference proxy object contains a "pointer" to the underlying .NET object, which continues to reside on the .NET side. A Java value proxy object contains a copy of the contents of the .NET object, which may or may not continue to reside on the .NET side. Reference and value proxies each have their own set of advantages.

When a reference proxy is passed as a parameter to the .NET side, or returned from the .NET side, only the reference is passed between .NET and Java. Since the reference is typically much smaller than the actual object, passing an object by reference is typically very fast. In addition, since the reference proxy points to a .NET object that continues to exist on the .NET side, if that .NET object is

updated, the updates are immediately accessible to the proxy object on the Java side. The disadvantage of reference proxies is that any access to data in the underlying object (e.g., field or method accesses) requires a round trip from the Java side to the .NET side (where the information resides) and back to the Java side.

When a value proxy is passed as a parameter to the .NET side, or returned from the .NET side, a copy of the object and its contents is passed between .NET and Java. Since the object itself is typically bigger than a reference, passing an object by value takes longer than passing it by reference. In addition, the value that is passed is a snapshot of the object taken at the time that it was passed. Since the passed object maintains no connection to the underlying .NET object, any updates to the underlying .NET object will not be reflected in the passed value proxy. The advantage of value proxies is that all data in the object is local to the Java side, and field accesses are very fast, since they do not require a round trip to the .NET side and back to get the data.

The choice of whether to use reference or value proxies depends on the desired semantics of the generated proxies as well as performance considerations. In general, one should use reference proxies, since they maintain the normal parameter-passing semantics of Java and C#. If a proxy is simply being used as a data transfer object, if there will be frequent accesses to its data, and if it is either relatively small or the frequency of accesses to data outweighs the time taken to transfer the object, make that proxy a value proxy.

Structure of value proxies

For Java-to-.NET projects, JNBridgePro supports two styles of value proxies, the *public/protected fields/properties* style, and the *directly mapped* style. A public/protected fields/properties value proxy contains copies of all the public/protected fields and properties of the underlying .NET object. For each public field or property X in the underlying .NET object, a public/protected fields/properties value proxy contains public methods of the form Get_X() and Set_X() allowing access to a virtual private field X containing a snapshot of the value returned by the field or property in the underlying .NET object at the time the proxy was returned from the .NET side. A Get_X() method takes no parameters and returns a value of the type of X. A Set_X() method takes a single parameter of the type of X, and returns no value. Note that public/protected fields/properties value proxies provide getters and setters for only unindexed public properties. A directly mapped value proxy contains the entire contents of the underlying .NET object translated into the equivalent Java object. For more information on directly mapped collection proxies, see "Directly-mapped collections," and "Other directly-mapped value objects," below.

Note that a value proxy object may contain members that are reference proxies. For example, if an object of class X (where X is a value proxy class) contains a field of type Y (where Y is a reference proxy class), values stored in that field will be reference proxies. Similarly, a reference proxy object may return (or take as parameters) value proxy objects.

If a .NET class is to have a value proxy generated for it, it must meet several requirements. If it is to be passed from the Java side to the .NET side as a parameter (or is to be assigned to the field of a reference proxy), the .NET class must have a public default constructor (that is, a public constructor taking no parameters). If it is to be used only in return values, the underlying .NET class need not have a public default constructor. In addition, any public properties to be passed in the value object must have matched pairs of public getter and setter methods. If the value is only being returned from methods or field accesses, it must have the public getter method but need not have the corresponding public setter method. If these rules are violated, an exception will be thrown when the proxy object is improperly used.

Note: There is one exception to the above requirement that the underlying .NET object have a public default constructor in order for its proxy to be passed as an argument in a call from the Java side (or to be assigned to a field or property from the Java side). .NET objects inheriting from System.ValueType (that is, structs) do not have an explicit default constructor, yet may be proxied as value proxies and passed from the Java side to the .NET side without causing an exception.

All value proxies generated by the proxy generation tool have a public default constructor (regardless of whether the underlying .NET class has one) which constructs an object of that class but does not initialize its data. If a value proxy explicitly constructed on the Java side through the **new** operator, it is the responsibility of the Java-side program to initialize the proxy's members through calls to setter methods.

All public/protected fields/properties-style value proxies have public setter methods corresponding to the public getter methods in the underlying .NET class's properties, even if the .NET class's underlying property does not have such a setter method. This is done to allow the properties in public/protected fields/properties-style value proxies to be initialized.

All value proxy classes inherit from, and implement, the same base classes and interfaces as the underlying .NET object. Value proxies have methods corresponding to each public method in the underlying .NET object. With the exception of static methods and of getter/setter methods corresponding to the underlying properties, calling these methods will cause a com.jnbridge.jnbcore.NotImplementedException to be thrown. The reason for this is that while the methods must exist to accommodate any interfaces the proxy implements or abstract classes the proxy inherits from, it is impossible to translate the semantics of the methods of the underlying .NET classes to Java. If Java-side equivalents of the underlying .NET methods are necessary, the developer should create a new class deriving from the value proxy, and override the methods in the new class. Calls to static methods of by-value proxy classes will be passed through to the underlying Java class, as will accesses to static fields and properties.

Consistency rules for value proxies

When value proxy classes are generated, the classes are checked against a number of consistency rules. If the rules are violated, changes may be made to the value/reference type assignments of the proxies, and the user is notified of the changes. The consistency rules (and the actions taken when they are violated) are as follows:

- Any class that inherits from a value proxy must also be a value proxy. If the derived class has been designated as a reference proxy, it is automatically changed to be a value proxy.
- If a class and its subclass are both value proxies, they must be of the same style. If they are not, the subclass's style is changed to be the same as the base class.
- An interface may be designated as a value proxy, but only if it is of the public/protected fields/properties style.
- Any class implementing a public/protected fields/properties-style value interface is a public/protected fields/properties-style value proxy. If it is not, it is changed to be a public/protected fields/properties-style value proxy (unless the implementing class is a directly mapped value object, in which case it is left alone).
- Any classes nested inside a value proxy class are also value proxy classes of the same style as the enclosing class. If they are designated as reference proxies, or if they are value proxies of a

different style, they are changed to be value proxies of the same style (unless the enclosing class is a directly mapped value object, in which case the nested classes are left alone).

Directly-mapped collections

In order to decrease the time required to access elements of a collection, JNBridgePro offers the ability to directly translate the contents of a .NET collection to a Java collection and vice versa. This means that, when a .NET collection object is returned from a method call, the result is translated into the equivalent Java collection, with its contents (which may be either reference or value proxies) residing locally on the Java side. Similarly, when a Java collection object is passed as a parameter, the object is translated to into the equivalent .NET collection, with its contents residing locally on the Java side.

By default, proxies for collection classes are generated as conventional reference proxies. To cause a .NET collection class to be directly mapped to a Java collection class, it is necessary to generate the proxy as "By value (mapped collection)" (see "Designating proxies as reference or value," above). Currently, the .NET classes System.Collections.ArrayList and System.Collections.Hashtable can be directly mapped.

A generated proxy for a directly mapped .NET collection class has the same name as the underlying .NET collection (e.g., the proxy class for System.Collections.Hashtable is also named System.Collections.Hashtable). Like other value proxies, it implements all interfaces and APIs as the underlying .NET collection, and inherits from the same superclass as the underlying .NET collection so that it can be passed or returned wherever an appropriate interface type or superclass type is expected. Also like other value proxies, calling a non-static method in the proxy, or any constructor other than the default constructor, causes a com.jnbridge.jnbcore.NotImplementedException to be thrown. A directly-mapped collection proxy differs from other proxies in that it has a field NativeImpl that contains the equivalent native Java collection with a copy of the contents of the underlying .NET object. For System.Collections.ArrayList, the NativeImpl field contains a Java object of type java.util.ArrayList. For System.Collections.Hashtable, NativeImpl contains a Java object of type java.util.Hashtable.

Below is an example of the use of directly mapped collections. Assume that System.Collections.Hashtable has been generated as a directly mapped value proxy class. Also assume that c is a reference proxy object with methods f() and g(), where f() returns a System.Collections.Hashtable and g() takes a System.Collections.Hashtable as a parameter.

```
System.Collections.Hashtable dotnet_hashtable = c.f();
java.util.Hashtable java_hashtable = dotnet_hashtable.NativeImpl;
// contents of java_hashtable can now be directly accessed
. . .
java.util.Hashtable new_java_hashtable
   = new java.util.Hashtable();
new_java_hashtable.put("a", "1");
new_java_hashtable.put("b", "2");
System.Collections.Hashtable new_dotnet_table = new System.Collections.Hashtable();
new_dotnet_table.NativeImpl = new_java_table;
c.g(new_dotnet_table);
```

Note that accesses of elements of a directly mapped proxy object are faster than accesses through a reference proxy object, but transferring the directly mapped object across the .NET/Java boundary takes longer than transferring the equivalent reference proxy object. It is up to the developer to weigh the benefits and disadvantages of each proxy type and to determine the appropriate type in each case.

There are a few conditions that must be observed when using directly mapped collection proxies:

- When passing a directly mapped collection object as a parameter from the Java side to the .NET side, the NativeImpl object must contain only objects that can be passed to the .NET side: primitives, strings, proxy objects (including directly mapped collections containing legal objects), and arrays containing legal objects. If any other objects are contained in the collection object, a SerializationException will be thrown.
- When a directly mapped collection containing primitives is passed to the Java side, the received collection will contain "boxed" primitives (see "Boxed primitive types," above). For example, an integer 1 passed in a collection will be received as a System.BoxedInt object containing the value 1. This is because Java collections cannot contain primitives.
- Objects passed or returned inside a directly mapped collection are mapped according to JNBridgePro's mapping rules. For example, strings and arrays are mapped between native .NET and native Java versions, and if an object is returned for which a generated proxy does not exist, it is returned as the nearest class in the superclass chain for which a generated proxy exists.
- ***Important note:*** .NET Hashtables allow null values (that is, the value half of a key/value pair can be null), but null values are not allowed in Java Hashtables. Consequently, when a .NET Hashtable is returned that contains null values, in the mapped Java Hashtable these null values will be translated to the singleton instance of the class com.jnbridge.jnbcore.NullValue. Similarly, when passing a Java Hashtable to .NET as a mapped Hashtable, a null value in a key/value pair can be passed by assigning NullValue to the value portion of the key/value pair. When the Hashtable is passed to the .NET side, the NullValue will be automatically translated as null. The singleton instance of NullValue can be obtained by calling NullValue.nullValue().
  Note: The NullValue object can only be used in this context. If it is used anywhere else, an exception will be thrown.

Other directly-mapped value objects (dates, decimals)

For convenience, one can automatically map between the Java and .NET data types representing dates, and between the Java and .NET data types representing extended-precision values. In particular, JNBridgePro can map between Java's java.util.Date class and .NET's System.DateTime class, and between Java's java.math.BigDecimial and java.math.BigInteger classes and .NET's System.Decimal class.

By default, Java-side proxies for DateTime and Decimal are generated as conventional reference proxies. To cause one of these .NET classes to be directly mapped to its corresponding Java class, it is necessary to generate the proxy as "By value (mapped)" (see "Designating proxies as reference or value," above).

A generated proxy for a directly mapped .NET class has the same name as the underlying .NET class (e.g., the proxy class for System.DateTime is also named System.DateTime). Like other value proxies, it implements all interfaces and APIs as the underlying .NET class, and inherits from the same superclass as the underlying .NET class so that it can be passed or returned wherever an appropriate interface type or superclass type is expected. Also like other value proxies, calling a non-static method in the proxy, or any constructor other than the default constructor, causes a com.jnbridge.jnbcore.NotImplementedException to be thrown. A directly-mapped value proxy differs from other proxies in that it has a field NativeImpl that contains the equivalent native Java value with a copy of the contents of the underlying .NET object. For System.DateTime, the NativeImpl field contains a Java object of type java.util.Date. For System.Decimal, NativeImpl contains a .NET object of type java.math.BigDecimal.

Below is an example of the use of directly mapped value objects. Assume that System.DateTime has been generated as a directly mapped value proxy class. Also assume that c is a reference proxy object with methods f() and g(), where f() returns a System.DateTime and g() takes a System.DateTime as a parameter.

```
System.DateTime dotnet_datetime = c.f();
java.util.Date java_date = dotnet_datetime.NativeImpl;
// contents of java_date can now be directly accessed
. . .
java.util.Date new_java_date
   = new java.util.Date(104, 10, 5, 11, 04, 34); // November 5, 2004, 11:04:34AM
System.DateTime  new_dotnet_datetime = new System.DateTime();
new_dotnet_datetime.NativeImpl = new_java_date;
c.g(new_dotnet_datetime);
```

Note that accesses of data inside a directly mapped proxy object are faster than accesses through a reference proxy object, but transferring the directly mapped object across the .NET/Java boundary takes longer than transferring the equivalent reference proxy object. It is up to the developer to weigh the benefits and disadvantages of each proxy type and to determine the appropriate type in each case.

Note: Java and .NET use differing time quanta for their Date/DateTime classes. This means that some accuracy in the microsecond/100-nanosecond range will be lost when passing .NET Dates to Java. It also means that, when passing a .NET DataTime around DateTime.MaxValue to Java and then returning the value, the returned value might be greater than DateTime.MaxValue and will cause an ArgumentOutOfRangeException to be thrown on the .NET side. Finally, it is possible to create Java Dates greater than DateTime.MaxValue; when these are passed from Java to .NET, an ArgumentOutOfRangeException will be thrown on the .NET side.

Transaction-enabled classes and support for transactions

Certain .NET APIs, particularly those connected with monitors or transactions, may depend on certain calls occurring in the same thread. If they do not, an exception may be thrown. For example, calls to an API to begin a transaction and to commit a transaction may need to be made from the same thread. This may make it problematic to generate proxies for such APIs and to use them from Java. The .NET side maintains a pool of threads listening for remoting calls from the Java side. Ordinarily, the first available .NET thread is chosen to execute the remoting call. In such situations, one cannot count on proxy calls from a Java thread to be handled by the same .NET thread each time.

In order to address this problem, JNBridgePro allows the user to designate proxy classes as *transaction-enabled*. All Java calls to transaction-enabled proxy classes, even to different transaction-enabled proxy classes, from the same Java thread are guaranteed to all be handled by the same .NET thread. Conversely, Java calls to transaction-enabled proxy classes from different Java threads are guaranteed to be handled by different .NET threads. This means, for example, that if all classes participating in a transaction have their proxies designated to be transaction-enabled, then Java calls to those proxies can be used to drive the transaction.

It should be noted that a transaction may not be explicitly visible to the Java caller; it may be buried inside the .NET code called from Java. In such cases, it is still necessary for the proxy classes from which the proxy operations are eventually called to be designated as transaction-enabled, if the transaction lasts across more than one Java call. (If the duration of the transaction is a single Java call, the proxy need not be designated transaction-enabled, since the single call is guaranteed to be handled by a single thread.) If the user calls non-transaction-enabled proxies, and a transaction- or monitor-related exception is thrown, it is likely that a transaction or monitor was attempted somewhere in the called .NET code; in such cases, designating transaction-enabled proxies may solve the problem.

If an interface is designated to be transaction-enabled, then any dynamically generated proxy classes implementing that interface are automatically transaction-enabled. Any statically generated proxy (that is, a proxy explicitly generated by the jnbproxy proxy generation tool) implementing a transaction-enabled interface is only transaction-enabled if the class itself is explicitly designated as transaction-enabled.

Proxy classes designated as pass-by-value cannot be transaction-enabled. If a pass-by-value proxy class is designated as transaction-enabled, that designation is ignored.

To designate a proxy class as transaction-enabled, see the section "Designating proxies as transaction-enabled," above.

Users should use the transaction-enabled feature sparingly, and only where necessary. While proxies unnecessarily designated as transaction-enabled will continue to work, transaction-enabled proxies incur a performance penalty, since the calls must pass through an additional thread management mechanism. In addition, each thread that calls a transaction-enabled proxy causes a .NET thread to be reserved for it on the .NET side. These threads take time to create, consume resources, and, unlike conventional threads in the .NET-side thread pool, are not recycled to handle calls from multiple Java-side threads.

Note that in earlier versions of JNBridgePro, transaction-enabled proxies were known as "thread-true." Their functionality is identical to what used to be known as "thread-true" proxies, and can be used wherever thread-true proxies were previously used.

Enabling and configuring cross-platform transactions

Java Enterprise Edition and .NET both support distributed transactions that support the X/Open standard for the two-phase commit protocol. However, because of different implementations, distributed transactions are not automatically cross-platform and transaction managers on the two platforms cannot automatically participate in global transaction processing.

JNBridgePro supports Java-to-.NET cross-platform transactions by extending an existing *implicit* transaction on the JEE side to a managed *explicit* transaction on the .NET side. JNBridgePro will automatically create and and manage a .NET System.CommittableTransaction that is associated with the thread in which Java-to-.NET calls execute on the .NET side. This transaction is dependent on the dominant transaction on the calling Java side and participates in the two-phase commit protocol managed by the JEE transaction manager or monitor.

A simple example would be an Entity EJB executing in a JEE container that manages an implicit transaction context in which the EJB executes. In this example, the Entity Bean is providing a persistence mechanism for customer data for a Java-based CRM system; however, it has become necessary to integrate a .NET customer billing application into the Entity Bean. Using JNBridgePro to create transaction-enabled Java proxies of the billing application's API, the Entity Bean can create and update customer data on both the Java CRM and the .NET billing application. On the Java side, data integrity is ensured by the JEE container-managed transaction. On the .NET side, data integrity is ensured by a JNBridgePro managed .NET transaction. If either side throws an exception during data processing, both transactions are rolled back. If no exception occurs and the two-phase commit protocol commences, if either the Java or .NET transaction managers call for a rollback during the prepare phase, both transactions are rolled back.

Enabling transactions

Enabling cross-platform transactions on the Java side is initialized with this method

```
static void com.jnbridge.transaction.JavaTM.enable(javax.transaction.Transaction transaction,
    int timeoutInSeconds)
```

where *timeoutInSeconds* is the timeout set for the container-managed transaction and *transaction* is the javax.transaction.Transaction associated with the container. This method not only ***must*** be called prior to any transaction-enabled Java-to-.NET calls, it ***must*** be called prior to any Java calls that invoke a resource that enlists in the container-managed transaction. All Java-to-.NET proxy methods and constructors called after cross-platform transactions have been enabled must be designated transaction-enabled.

Warning: If a Java-to-.NET class that is *not* designated transaction-enabled is constructed or its methods invoked after cross-platform transactions are enabled, a JNBTransactionException will be thrown. *This will cause an immediate rollback on both platforms.*

Obtaining the the Java side container-managed transaction

In order to enable cross-platform transactions using the JavaTM.enable() method, it is necessary to obtain the implicit container-managed javax.transaction.Transaction object so it can be used as the *transaction* argument to the enable() call. Obtaining the transaction object is implementation specific and peculiar to the vendor of the JEE application server. Generally, the developer must obtain the javax.transaction.TransactionManager object using a vendor specific static helper method and call TransactionManager.getTransaction(). Some JEE application server vendors provide helper methods

that return the current transaction. The following code example uses Oracle WebLogic (version 10gR3).

```
public String ejbCreate(String visitorName) throws CreateException
{
   // obtain the container's implicit transaction using WebLogic static helper class
   Transaction trans = TransactionHelper.getTransactionHelper().getTransaction();
   try
   {
      com.jnbridge.transaction.JavaTM.enable(trans, 600);
   }
   catch (Exception ex) // JNBTransactionException or SystemException
   {
      throw new CreateException(
         "Hey! Check your configuration-does this EJB require transactions" );
   }
   // EJB persistence calls
   setVisitorName(visitorName);
   setVisitNumber(1);
   // JNBridgePro Java-to-.NET calls to transaction-enabled proxy
   UpdateDB dotNetDB = new UpdateDB();
   dotNetDB.createConnection();
   dotNetDB.insertNewCustomer(this.getVisitorName(), this.getVisitNumber());
   dotNetDB.closeConnection();
   return null;
}
```

Coding guidelines

In order to ensure proper function of cross-platform transactions, please follow these coding guidelines.

- Always call com.jnbridge.transaction.JavaTM.enable() before any other method calls that are local or Java-to-.NET.
- Always call com.jnbridge.transaction.JavaTM.enable() in the same EJB method as all the Java-to-.NET calls. Transaction-enabled JNBridgePro proxies are also *thread-true*; the enable() call must happen in the same Java thread as the subsequent Java-to-.NET calls.
- Never catch exceptions thrown by the Java-to-.NET calls. If an exception must be caught, either re-throw it or throw another exception in the try block. Exceptions are caught by the EJB's container notifying the transaction manager to abort the transaction and initiate a rollback before the two-phase commit commences.
- Using explicit JEE transactions instead of implicit container-managed transactions is not permitted.

Use of JNBridgePro for bi-directional interoperability

It is possible to construct programs that contain both .NET-to-Java and Java-to-.NET calls; that is, the Java and .NET sides are each both client and server. To get this full two-way interoperability, one must make sure that the .NET side and Java side each start a server, as described elsewhere in this document. (On the .NET side, call DotNetSide.startDotNetSide(), and on the Java side, call JNBMain.start().) In addition, the Java side must initialize the Java-side client by calling DotNetSide.init().

Any Java program taking part in two-way communication must have jnbcore.jar and bcel-*6.n.m*.jar in its classpath.

The Java-side and .NET-side configuration files mush have both client and server configuration information. If configuring the .NET side through an application configuration file, the application file must have both dotNetToJavaConfig and javaToDotNetConfig elements. See the sections "Configuring the .NET side" under both "Using JNBridgePro for .NET-to-Java Calls" and "Using JNBridgePro for Java-to-.NET Calls" for more information.

If configuring the .NET side programmatically, one must use the version of JNBRemotingConfiguration.specifyRemotingConfiguration() that defines both remote and local information. See "Configuring the .NET side" for more information.

A sample jnbcore.properties Java-side configuration file is below. Note the presence of both javaSide.* and dotNetSide.* properties. Edit the various properties as appropriate.

```
# Java-side (.NET-to-Java) properties
javaSide.serverType=tcp
javaSide.workers=5
javaSide.timeout=10000
javaSide.port=8085
# .NET-side (Java-to-.NET) properties
dotNetSide.serverType=tcp
dotNetSide.host=localhost
dotNetSide.port=8086
```

Bi-directional shared-memory communications

When using the shared-memory communications channel for bi-directional communications, special attention must be paid to the way in which JNBridgePro is configured. When bidirectional shared-memory communication is used, the manner of configuration depends on whether the program is started on the .NET side, or on the Java side.

If the program is started on the Java side, follow the instructions for configuring the Java side for shared-memory communication as described previously in the *Users' Guide*. Nothing additional needs to be done. All the configuration information is placed in Java-side configuration file jnbcore.properties, or is supplied to the Java side from the command line on startup.

If the program is started on the .NET side, all the configuration information is supplied in the .NET-side application configuration file, or programmatically through the specifyRemotingConfiguration() API. Note that all configuration will be supplied on the .NET side. There is no need to supply Java-side configuration information.

To specify the bi-directional shared memory configuration programmatically, one of the following new variants of com.jnbridge.jnbcore.JNBRemotingConfiguration.specifyRemotingConfiguration() must be used:

- specifyRemotingConfiguration(JavaScheme *remoteProtocol*, String *jvmLocation*, String *jnbcoreLocation*, String *bcelLocation*, String *classpath*, String[] *jvmOptions*, String *javaEntryLocation*)

  is used to specify bi-directional shared-memory communication where the program starts on the .NET side. *remoteProtocol* must be JavaScheme.sharedmem, or an exception will be thrown. *jvmLocation* is the full path to the file jvm.dll which implements the Java Virtual Machine and which is typically distributed with a JDK (Java Development Kit) or JRE (Java Runtime Environment). *jnbcoreLocation* is the full path to jnbcore.jar. *bcelLocation* is the full path to bcel-*6.n.m*.jar. *classpath* is the semicolon-separated classpath to be used by the Java side. *jvmOptions* is a string array containing any additional options to be supplied to the JVM upon startup. For example, if you wish to specify that the maximum side of the JVM's memory allocation pool should be 80 megabytes, *jvmOptions* should contain the string "-Xmx80m". *javaEntryLocation* is the full path to the file JNBJavaEntry_x86.dll or JNBJavaEntry_x64.dll. If you want to run the application under both 32-bit and 64-bit applications, *javaEntryLocation* should be the full path to a folder containing both JNBJavaEntry_x86.dll *and* JNBJavaEntry_x64.dll.

- specifyRemotingConfiguration(JavaScheme *remoteProtocol*, String *jvmLocation*, String *jnbcoreLocation*, String *bcelLocation*, String *classpath*, String *javaEntryLocation*)

  is a convenience version of the previous version of specifyRemotingConfiguration(), used when bi-directional shared-memory communication starting on the .NET side is to be used and there are no additional JVM options to be supplied. It is equivalent to calling

  specifyRemotingConfiguration(*remoteProtocol*, *jvmLocation*, *jnbcoreLocation*, *bcelLocation*, *classpath*, new String[0], *javaEntryLocation*);

When configuring JNBridgePro using the application configuration file, use one of the following configuration elements:

```
<dotNetToJavaConfig scheme="sharedmem"
                    jvm="full path to jvm.dll in JDK or JRE"
                    jnbcore="full path to jnbcore.jar"
                    bcel="full path to bcel-6.n.m.jar"
                    classpath="semicolon-separated Java classpath"
                    javaEntry="full path to JNBJavaEntry.dll" />
```

When using shared memory, additional options (for example, the maximum size of the memory allocation pool), they can be supplied through additional `jvmOptions` attributes. For example, to specify that the maximum size of the memory allocation pool should be 80 megabytes, use the following configuration element:

```
<dotNetToJavaConfig scheme="sharedmem"
                    jvm="full path to jvm.dll in JDK or JRE"
                    jnbcore="full path to jnbcore.jar"
                    bcel="full path to bcel-6.n.m.jar"
                    classpath="semicolon-separated Java classpath"
                    javaEntry="full path to JNBJavaEntry.dll"
                    jvmOptions.0="-Xmx80m"/>
```

All `jvmOptions` attributes must be of the form `jvmOptions.`*`n`*`="value"` where *`n`* can be any unique string, but is typically an integer in a sequence starting with 0.

Note that `javaEntry` above should be the path to the JNBJavaEntry_x86.dll (if the application will only run as 32-bit), JNBJavaEntry_x64.dll (if the application will only run as 64-bit), or to a folder

containing both JNBJavaEntry_x86.dll and JNBJavaEntry_x64.dll (if the application can run as either 32-bit or 64-bit).

In addition to the above, it is necessary to supply the list of assemblies that will be accessed from the Java side. To do this, you can use application configuration files with the <assemblyList>. Add the following elements inside the <jnbridge> section of the application configuration file:

```
<assemblyList>
   <assembly file="path to assembly or fully qualified name"/>
   ...
</assemblyList>
```

Alternatively, add a call to

Com.jnbridge.jnbcore.DotNetSide.startDotNetSide(string[] *assemblyList*)

where *assemblyList* is an array of strings representing the full or relative paths of the assemblies in which the .NET side should search for .NET types being accessed from the Java side.If an assembly is in the Global Assembly Cache (GAC), it can be specified using its fully-qualified name. For example, the .NET 2.0 version of the assembly System.Windows.Forms.dll may be specified as

"System.Windows.Forms, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089"

To access all the assemblies in a folder, simply supply the full path of the folder.

Types defined in the assembly mscorlib.dll are always searched and these assemblies need not be listed in the *assemblyList*.

Appendix: jnbproxy.config

Prior to JNBridgePro v2.1, the .NET side was configured through the use of a special configuration file, jnbproxy.config. Starting with v2.1, we recommend configuring JNBridgePro through the application's configuration file or programmatically in the application itself, and use of jnbproxy.config has been deprecated. However, use of jnbproxy.config is still supported for backward compatibility,. To assist users who may be migrating from earlier versions of JNBridgePro, we include details on jnbproxy.config in this appendix.

jnbproxy.config for .NET-to-Java calls

The .NET-side configuration file, jnbproxy.config, includes configuration for both .NET-side clients (for .NET-to-Java calls) and .NET-side servers (for Java-to-.NET calls). Prototype .NET-side configuration files are included in the JNBridgePro v12.0 installation folder: jnbproxy_tcp.config is a prototype configuration file for tcp/binary communication, and jnbproxy_sharedmem.config is a prototype configuration file for shared-memory communication.

The tcp/binary configuration file contains the following configuration elements for setting up a .NET-side client:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <client>
        <wellknown type="com.jnbridge.jnbcore.JNBDispatcher,JNBShare"
          url="jtcp://localhost:8085/JNBDispatcher" />
      </client>
    <channels>
    <channel type="com.jnbridge.jnbproxy.JNBBinaryChannel, JNBShare,
      Version=12.10.0.0, Culture=neutral, PublicKeyToken=b18a44fb28aea122" "/>
    </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

Note that for historical reasons, for the 4.8-targeted version of JNBridgePro being used, the <channel> element above should refer to "JNBShare, Version=12.10.0.0," not "12.0.0.0".

The shared-memory configuration file contains the following configuration elements for setting up a .NET-side client:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <client>
        <wellknown type="com.jnbridge.jnbcore.JNBDispatcher,JNBShare"
          url="sharedmem:///JNBDispatcher" />
      </client>
    <channels>
       <channel type="com.jnbridge.jnbproxy.JNBSharedMemChannel, JNBShare,
         Version=12.10.0.0, Culture=neutral, PublicKeyToken=b18a44fb28aea122"
          jvm="location of jvm.dll distributed with JDK or JRE"
          jnbcore="location of jnbcore.jar"
          bcel="location of bcel-6.n.m.jar"
          classpath="semicolon-separated classpath"/>
    </channels>
    </application>
```

```
  </system.runtime.remoting>
</configuration>
```

Note that for historical reasons, for the 4.8-targeted version of JNBridgePro being used, the <channel> element above should refer to "JNBShare, Version=12.10.0.0," not "12.0.0.0".

Note that the URL element in the wellknown tag specifies the means by which the .NET side communicates with the Java side. The protocol (jtcp for binary/tcp communication, or sharedmem for shared-memory communication) specifies the communications protocol that will be used, the host name indicates the host on which the Java side resides (it can be a host name or an IP address, and can be "localhost" if the Java side is on the same machine as the .NET side), and the port number indicates the port on which the Java side will be listening for requests. Note that when shared-memory communication is used, the URL has no port or host name specified.

When configuring shared-memory communication, you must fill in the following values:

- *jvm*: the full path of the jvm.dll file that implements the Java virtual machine. jvm.dll is typically distributed as part of a Java Development Kit (JDK) or Java Runtime Environment (JRE).
- *jnbcore*: the full path of the jnbcore.jar file
- *bcel*: the full path of bcel-*6.n.m*.jar
- *classpath*: a semicolon-separated list of folders and files representing the classpath

When using shared memory, additional options (for example, the maximum size of the memory allocation pool), they can be supplied through additional `jvmOptions` attributes. For example, to specify that the maximum size of the memory allocation pool should be 80 megabytes, use the following configuration element:

```
<channel type="com.jnbridge.jnbproxy.JNBSharedMemChannel, JNBShare,
  Version=12.10.0.0, Culture=neutral, PublicKeyToken=b18a44fb28aea122"
   jvm="location of jvm.dll distributed with JDK or JRE"
   jnbcore="location of jnbcore.jar"
   bcel="location of bcel-6.n.m.jar"
   classpath="semicolon-separated classpath"
   jvmOptions.0="-Xmx80m"/>
```

Note that for historical reasons, for the 4.8-targeted version of JNBridgePro being used, the <channel> definition above should refer to "JNBShare, Version=12.10.0.0," not "12.0.0.0".

All `jvmOptions` attributes must be of the form `jvmOptions.`*`n`*`="value"` where *`n`* can be any unique string, but is typically an integer in a sequence starting with 0.

Aside from the protocol, host, and port values in the URL element of the <wellknown> tag, or the jvm/jnbcore/bcel/classpath/jvmOptions elements in the shared-memory <channel> tag, do not edit any other portion of the configuration files listed above. If you do, JNBridgePro communication between .NET and Java may stop working.

The jnbproxy.config configuration file must be placed according to the rules described in the section "Placement of configuration file jnbproxy.config."

jnbproxy.config for Java-to-.NET calls

The .NET-side configuration file, jnbproxy.config, includes configuration for both .NET-side clients (for .NET-to-Java calls) and .NET-side servers (for Java-to-.NET calls). Prototype .NET-side

configuration files are included in the JNBridgePro installation folder: jnbproxy_tcp.config is a prototype configuration file for tcp/binary communication.

The tcp/binary configuration file contains the following configuration elements for setting up a .NET-side server:

```
<configuration>
  <system.runtime.remoting>
    <application>
    <channels>
    <channel type="com.jnbridge.jnbproxy.JNBBinaryChannel,JNBShare" port="8086" />
    </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

Note that the port element in the channel tag specifies the port on which the .NET-side server will be listening for requests. It should be edited to specify the port on which the .NET-side server will be listening if this port is different from 8086.

Aside from the port values, do not edit any other portion of the configuration files listed above. If you do, JNBridgePro communication between .NET and Java may stop working.

The jnbproxy.config configuration file, must be placed according to the rules described in the section "Placement of configuration file jnbproxy.config."

Placement of configuration file jnbproxy.config

JNBridgePro searches for the configuration file `jnbproxy.config` in the following folders:

- It first looks in the folder where `jnbshare.dll` is located. Note that this is the folder from which jnbshare.dll is executed, or the folder from which a shadow-copy is done, in the case of ASP.NET or other applications that shadow-copy DLLs into another framework before executing.
- If it does not find it 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. (This is done for purposes of backward compatibility with older versions of JNBridgePro, but this is no longer necessary.)
- If it does not find it there, it looks in the folder `<System-drive>:\` (the root folder on the drive on which the system is installed; typically C, but not always). (This is done for purposes of backward compatibility with older versions of JNBridgePro, but this is no longer necessary.)

jnbproxy.config for bidirectional interoperability

The Java-side and .NET-side configuration files must have both client and server configuration information. The file jnbproxy_tcp.config contains examples of two-way configuration for tcp/binary.

jnbproxy_tcp.config is as follows. To use two-way tcp/binary communication, copy the file to jnbproxy.config in the appropriate location, and edit the indicated hostname and ports.

```
<configuration>
  <system.runtime.remoting>
    <application>
      <client>
```

```
      <!-- Configuration for .NET-to-Java -->
      <!-- Edit host name and port only -->
          <wellknown type="com.jnbridge.jnbcore.JNBDispatcher,JNBShare"
            url="jtcp://localhost:8085/JNBDispatcher" />
        </client>
      <channels>
      <!-- Configuration for Java-to-.NET -->
      <!-- Edit port only -->
      <channel type="com.jnbridge.jnbproxy.JNBBinaryChannel,JNBShare" port="8086" />
      </channels>
      </application>
    </system.runtime.remoting>
  </configuration>
```