Demo: Controlling .NET Windows Forms from a Java Application

Version 11.0

jnbridge.com

JNBridge, LLC
jnbridge.com

COPYRIGHT © 2001–2021 JNBridge, LLC. All rights reserved.

JNBridge is a registered trademark and JNBridgePro and the JNBridge logo are trademarks of JNBridge, LLC.

Java is a registered trademark of Oracle and/or its affiliates. Microsoft and Visual Studio are trademarks or registered trademarks of Microsoft Corporation in the United States and other countries.

All other marks are the property of their respective owners.

September 27, 2021

Introduction

This document shows how JNBridgePro can be used to construct a Java application that calls .NET classes, particularly .NET Windows Forms. The reader will learn how to generate Java proxies that call the .NET classes, create Java code that calls the proxies and, indirectly, the corresponding .NET classes, and set up and run the code. The user will also learn how to use callbacks in the context of Java-to-.NET projects.

In the example, JNBridgePro is used to allow a Swing-based Java application to call and manipulate .NET-based Windows Forms. There are a number of reasons one might want to do such a thing. The developer may have a legacy Java application, and may wish to add migrate it gradually into the .NET platform without having to rewrite it all at once. In such a case, it may be preferable to create new functionality on the .NET platform while leaving the Java code alone, or to rewrite small amounts of existing functionality in .NET while not touching the remainder of the code. Another possible reason is that the developer may wish to have the Java application make use of a .NET control whose equivalent does not exist in Java.

In this example, we assume two existing .NET Windows Forms classes, SwingInterop.Form1 and SwingInterop.Form2. We will create a Java application that will, in response to button-clicks on its Swing-based user interface, pop up instances of the .NET-based Form1 and Form2, then display in a Swing window the data entered in the .NET forms through a callback registered with the .NET code.

Generating the proxies

While this example uses the standalone proxy generation tool, you can also use the Visual Studio plug-in, and the example figures will look very much the same.

The first step in the process is to generate proxies for SwingInterop.Form1 and Form2, and for their supporting classes. Start by launching JNBProxy, the GUI-based proxy generator. When it is launched, you will first see a *launch form* (Figure 1). Choose “Create New Java→.NET Project, and you will see JNBProxy’s main form (Figure 2), configured for a Java-to-.NET project.

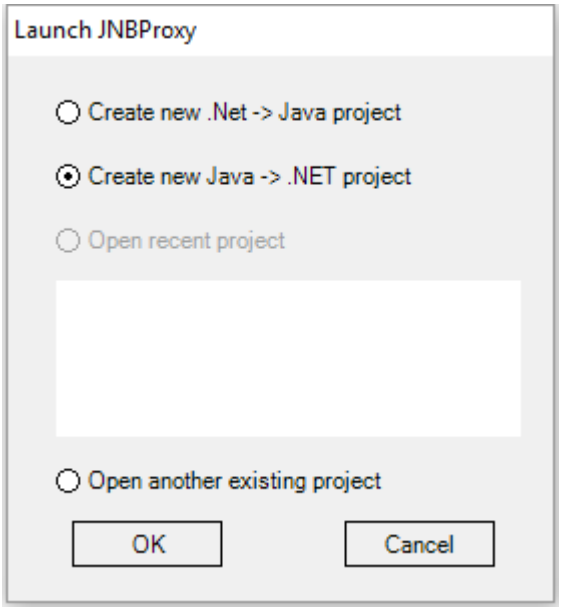

Figure 1. JNBProxy launch form

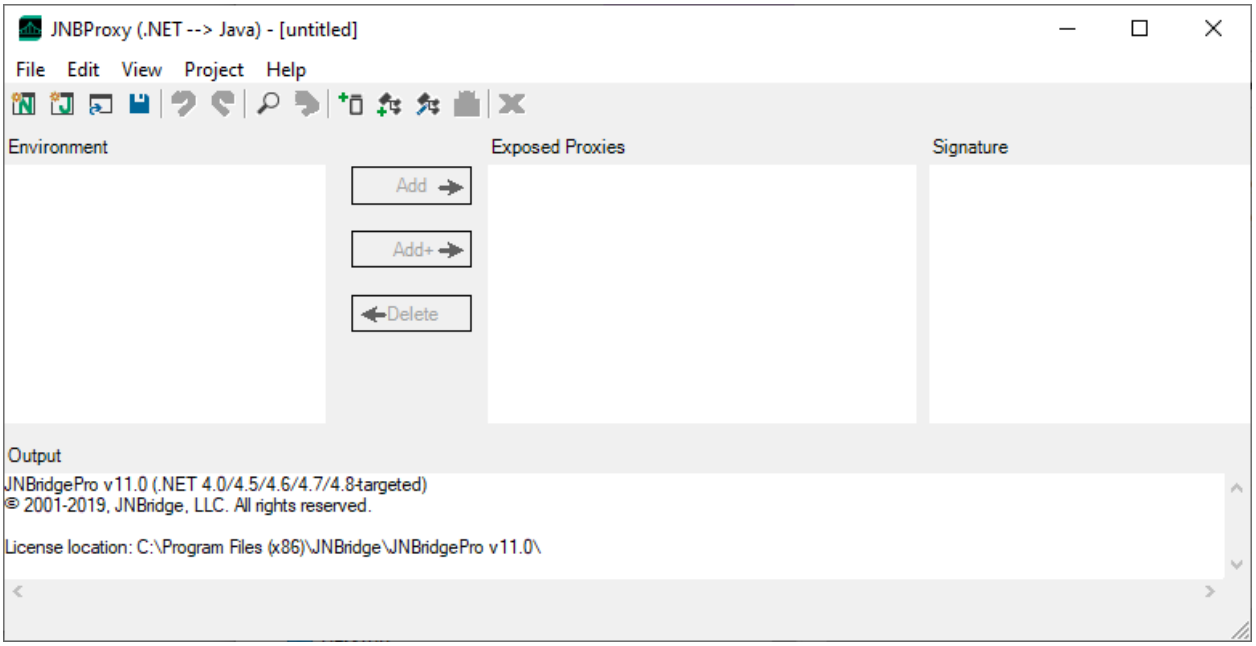

Figure 2. JNBProxy

Next, add the assemblies SwingInterop.dll and System.Windows.Forms.dll to the assembly list to be searched by JNBProxy. (We will be calling methods that are not defined in SwingInterop.Form1 and Form2, but rather in their superclass System.Windows.Forms.Form, which is defined in System.Windows.Forms.dll.) Use the menu command **Project→Edit Assembly List...**. The **Edit Assembly List** dialog box will come up, and clicking on the **Add...** button will bring up a dialog that will allow the user to indicate the paths of SwingInterop.dll (Figure 3).

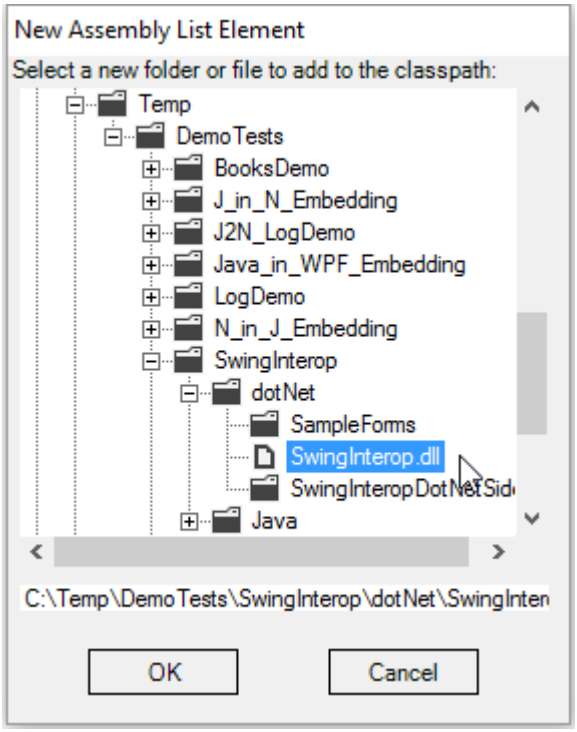

Figure 3. Adding a new assembly list element

System.Windows.Forms.dll is in the Global Assembly Cache (GAC). Add it to the assembly list by clicking on the **Add From GAC...** button, and selecting the System.Windows.Forms.dll from the displayed list (Figure 4). If you have more than one version of the .NET Framework installed on your machine, you may have more than one version of System.Windows.Forms.dll in the GAC; select the appropriate one.

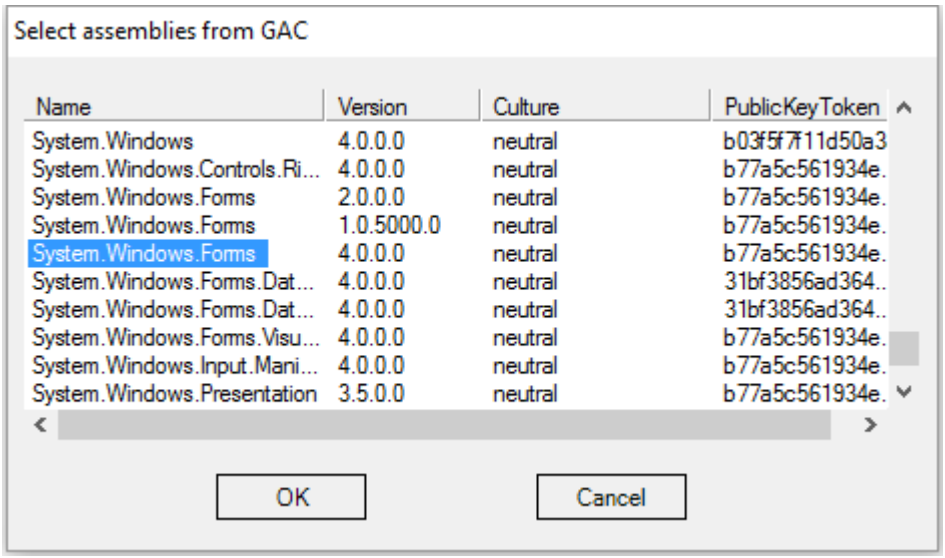

Figure 4. Selecting an assembly from the GAC

When all the necessary elements of the classpath are added, the **Edit Assembly List** dialog should contain information similar to that shown in Figure 5.

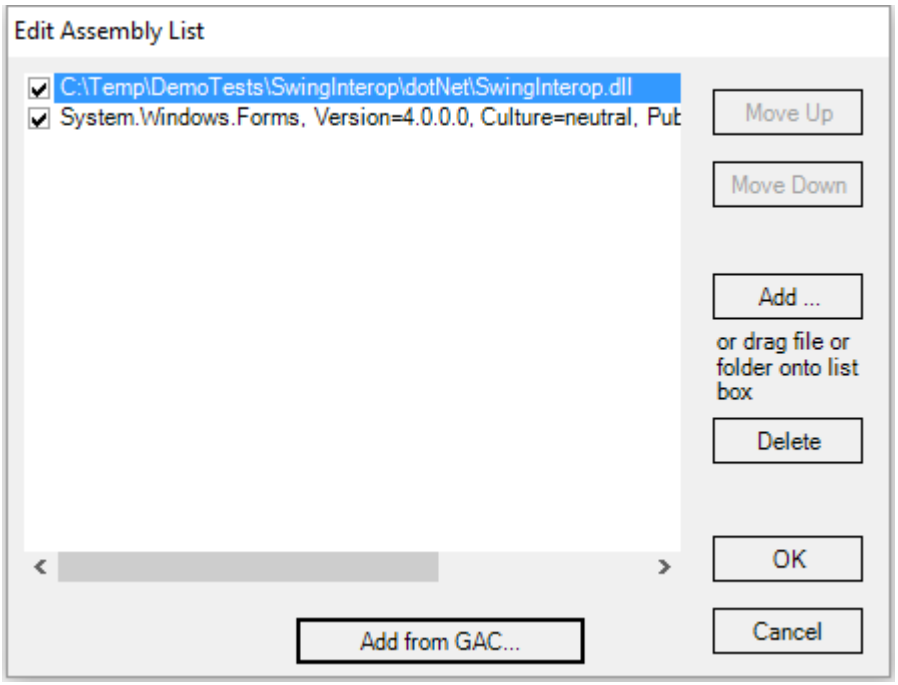

Figure 5. After creating assembly list

The next step is to load the classes Form1, Form2, and JavaWindowEventArgs, plus the supporting classes. Use the menu command **Project→Add Classes from Assembly List...** and enter the fully qualified class names SwingInterop.Form1, SwingInterop.Form2, and SwingInterop.JavaWindowEventArgs, making sure that the "Include supporting classes" checkbox is checked for each (Figure 6). Finally, add System.Windows.Forms.UnsafeNativeMethods (also with "Include supporting classes" checked).

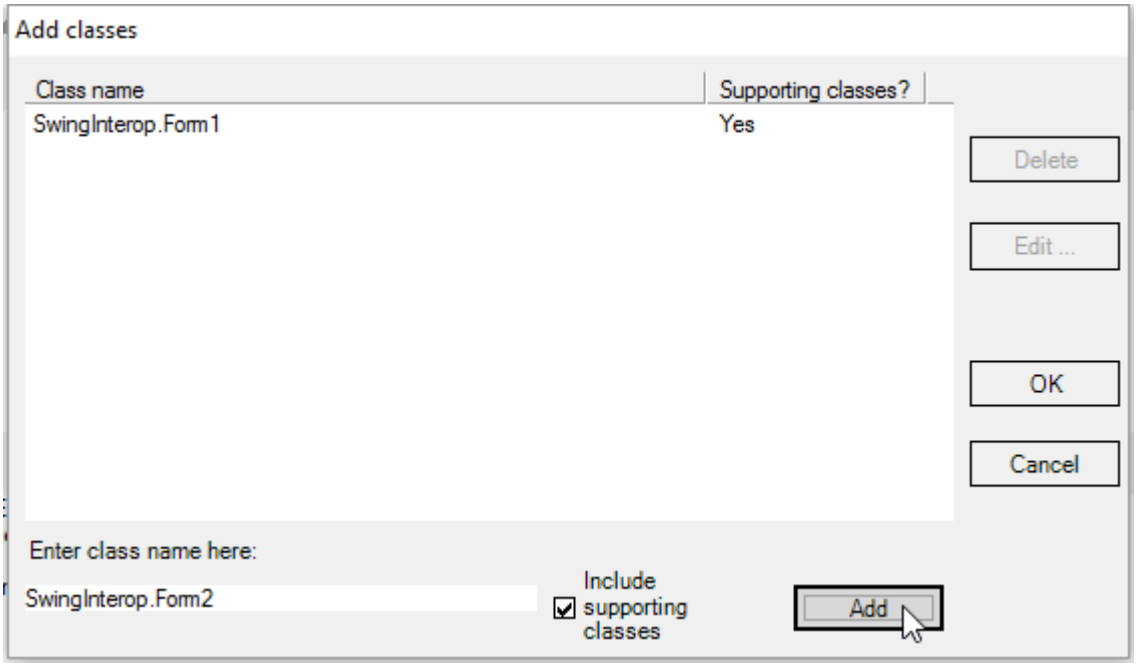

Figure 6. Adding a class from the classpath

Loading the classes may take a few seconds. Progress will be shown in the output pane in the bottom of the window, and in the progress bar. When completed, Form1, Form2, and all their supporting classes will be displayed in the Environment pane on the upper left of the JNBProxy window (Figure 7). Note that JNBProxy will warn us that we are missing a number of classes. Since we are not going to use these capabilities, we can safely ignore this warning.

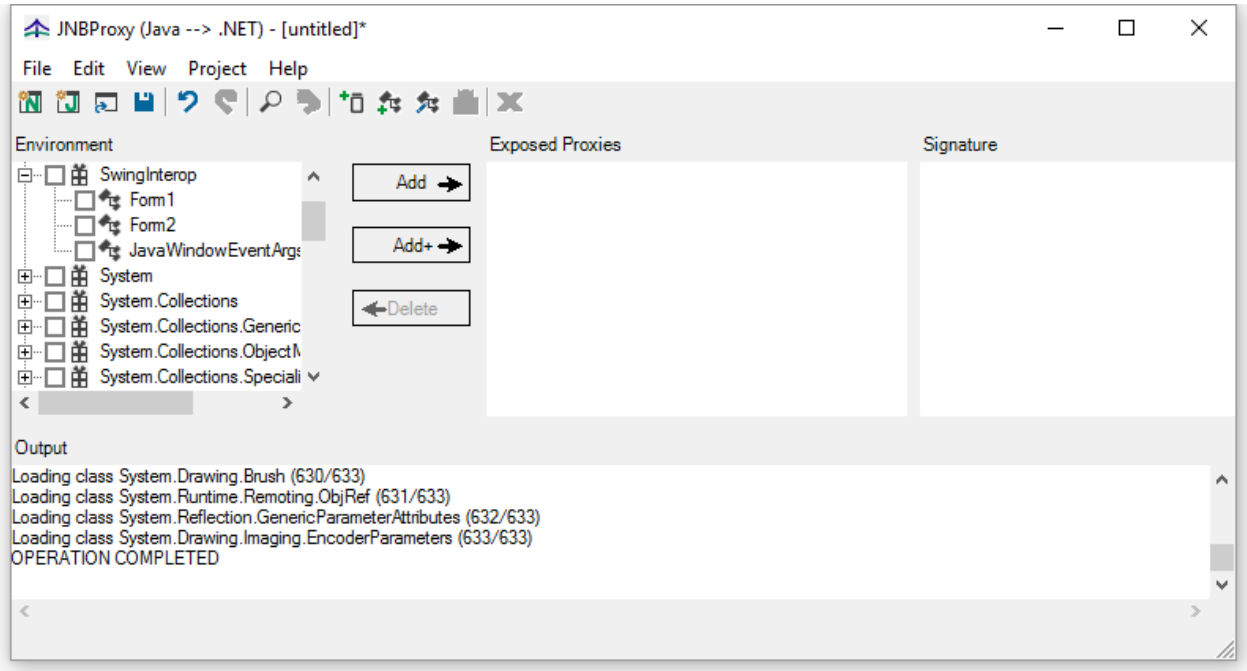

Figure 7. After adding classes

We wish to generate proxies for all these classes, so when all the classes have been loaded into the environment, make sure that each class in the tree view has a check mark next to it. Quick ways to do this include clicking on the check box next to each package name, or simply by selecting the menu command **Edit→Check All in Environment**. Once each class has been checked, click on the **Add** button to add each checked class to the list of proxies to be exposed. These will be shown in the Exposed Proxies pane (Figure 8).

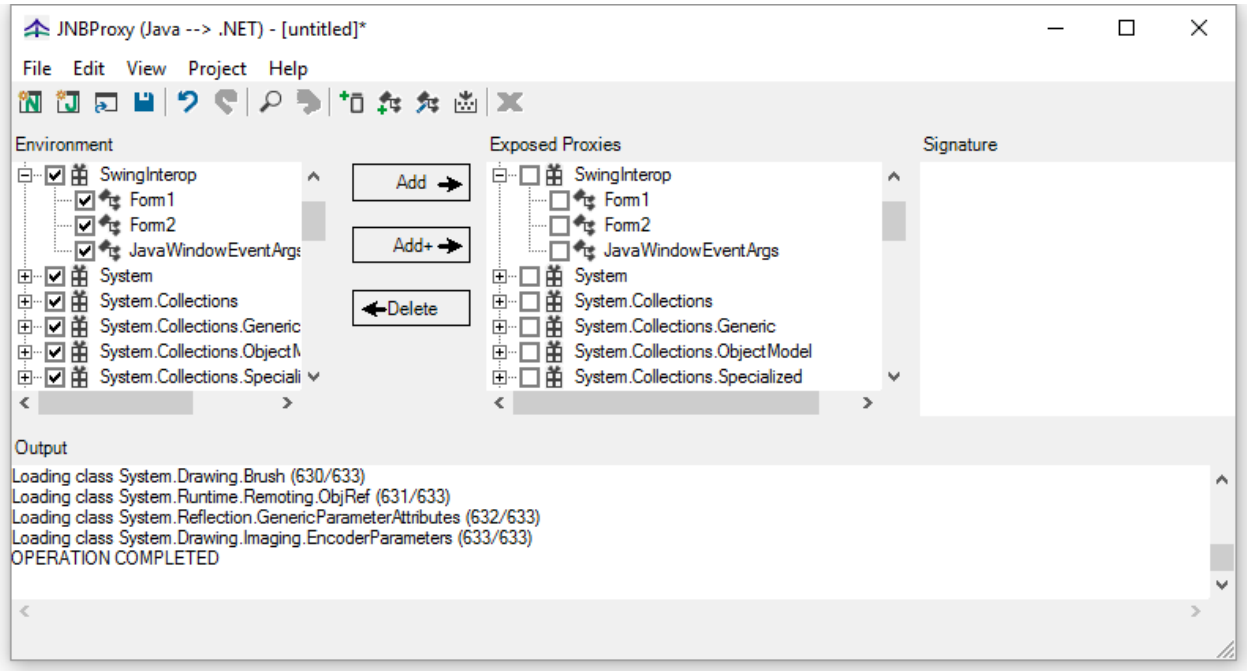

Figure 8. After adding classes to Exposed Proxies pane

We are now ready to generate the proxies. Select the **Project→Build...** menu command, and choose a name and location for the Java archive (.jar) file that will contain the generated proxies. The proxy generation process may take a few minutes, and progress and other information will be indicated in the Output pane. In this example, we will call the generated proxy assembly sampleForms.jar.

Using the proxies

Java side

Now that the proxies have been generated, we can use them to access .NET classes from Java. Examine the file MainForm.java, which defines a Java application that creates a Swing-based window with one text box and two buttons (Figure 9).

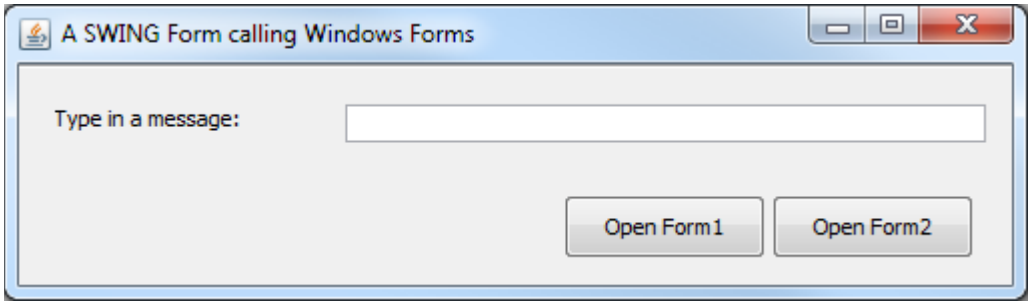

Figure 9. Sample Java application

When the "Open Form1" button is clicked, an instance of the .NET-based Form1 is displayed as a modal dialog box, displaying the contents of the text box. An instance of a callback object,

MessageWindowListener, is created and registered with the ButtonClicked event in form1. One can type a new message into Form1; when it is dismissed, the event is fired and MessageWindowListener is invoked, causing a message window to be displayed with the string that was entered in form1.

Clicking on "Open Form1" causes the actionPerformed() listener action inside the getJButton() method to be performed. actionPerformed() contains the following code:

```
// Open the form in modal mode
Form1 form1 = new Form1();
form1.Set_Message(jTextField.getText());
MessageWindowListener mwl = new MessageWindowListener(jTextField);
form1.add_ButtonClicked(mwl);
form1.ShowDialog();
form1.dotNetDispose();
```

Form1 is the proxy class representing SwingInterop.Form1 on the .NET side. The first line above creates a new instance of Form1, but doesn't display it. The second line sets the Message property in form1 with the value in the Java application's text field. This message will be displayed when form1 is displayed. The third and fourth lines create and register the Java-side callback object as a handler for the ButtonClicked event. The fifth line calls form1's ShowDialog() method, which causes the form to be displayed as a modal dialog box. Execution of the Java application is now suspended until the dialog box is dismissed and the ShowDialog() method returns. Once form1 is dismissed, the ButtonClick event is fired and the Invoke() method in the callback object is executed. The message entered in form1 is passed to Invoke() and Invoke() displays a message window with the message string. Finally, the .NET-side Form1 object is released. It is safe to wait until the Java side garbage-collects the proxy object, but it is efficient to dispose of it as soon as it is no longer being used.

When the "Open Form2" button is clicked, the .NET-based Form2 is displayed as a non-modal dialog box. Form2 is displayed until it is dismissed, and in the meantime the Java application can process additional input. Clicking on the button causes the actionPerformed() listener action inside the getJButton1() method to be performed. This second actionPerformed() method contains the following code:

```
// Open the second form, but not in modal mode
Form2.showNonModal();
```

Form2 is the proxy class for the .NET-based SwingInterop.Form2. Calling showNonModal() causes an instance of Form2 to be displayed as a non-modal dialog and the call immediately returns. The form will remain displayed until it is dismissed.

Finally, the main() method starts with the following code:

```
// initialize the Java-side client
DotNetSide.init(args[0]);
```

This call configures the Java-side runtime components of JNBridgePro. The command-line argument will be the path of the Java-side configuration file.

Note the following items of interest in the code above:

- .NET objects are instantiated from Java code simply by instantiating the corresponding proxy object.

- .NET objects' methods are called simply by calling the corresponding method in the proxy object. To call a static .NET method, simply call the corresponding method on the proxy class (e.g., Form2.showNonModal()).
- The MessageWindowListener object is recognized as a Java-side callback object because it implements the EventHandler proxy interface. Because it is a callback connected with Windows forms and Swing windows, it is also designated as an asynchronous callback object by implementing the AsyncCallback marker interface.
- The MessageWindowListener callback object is registered as an event handler for the ButtonClicked event by passing it as an argument to form1.add_ButtonClick().
- One can release the .NET object underlying the Java-side proxy by calling the dotNetDispose() method on the corresponding proxy object. The result is to make the underlying .NET object eligible for garbage collection. After dotNetDispose() is called, one can no longer use the proxy object.
- The method showDialog() was not defined in the class Form1, but rather in its superclass System.Windows.Forms.Form. This is why we generated proxies with all of Form1's supporting classes. If we did not have a proxy for the superclass Form, we would not have been able to call showDialog().
- Every Java-side client program using JNBridgePro to call .NET code must call DotNetSide.init() before any calls to the .NET side are performed.

To compile the Java code and prepare it for execution, copy the following files from the JNBridgePro installation folder into the Java\SampleClient folder in the demo:

- jnbcore.jar (the Java-side runtime component)
- jnbcore_tcp.properties (the Java-side configuration component)
- bcel-5.1-jnbridge.jar (a runtime library supporting the dynamic generation of proxy classes)

Also, move SampleForms.jar from wherever it was built, into the Java\SampleClient folder.

Make sure that you have installed a JDK (Java Development Kit). Once these files are copied, open a command-line window and compile the Java code by executing the following command:

```
javac -classpath ".;SampleForms.jar;jnbcore.jar;bcel-5.1-jnbridge.jar" MainForm.java
```

.NET side

Note the call to Form2.showNonModal() in the Java-side code. This method had to be specially defined on the .NET side to allow non-modal dialog boxes. The .NET file Form2.cs contains the following C# code defining showNonModal():

```
public static void showNonModal()
{
      ThreadStart ts = new ThreadStart(Invoke);
      Thread t = new Thread(ts);
      t.Start();
}

private static Form instance;
private static void Invoke()
```

```
{
        instance = new Form2();
        System.Windows.Forms.Application.Run(instance);
}
```

This code creates a new thread and runs an instance of Form2 in that thread.

To access .NET-side code from Java, one has to create a JNBridgePro .NET-side server to accept requests from the Java side and to manage the .NET objects. The server can be fairly simple. The main procedure in the file Class1.cs is all that is required:

```
static void Main(string[] args)
{
        // specify the assemblies we'll be accessing from Java
        string[] assemblies = {
                "SwingInterop.dll",
                "System.Windows.Forms, Version=1.0.5000.0, " +
                        "Culture=neutral, PublicKeyToken=b77a5c561934e089"
                  };
        // need to initialize DotNetSide with the names of all the
        // assemblies
        // from which you will be referencing classes
        DotNetSide.startDotNetSide(assemblies);
        Console.WriteLine("Hit <return> to exit");
        Console.ReadLine();
        DotNetSide.stopDotNetSide();  // end it (optional)
}
```

Note that all that needs to be done is to call DotNetSide.startDotNetSide() and supply it with a list of all the assemblies that we will be using. At that point, we just have to wait until the program is finished, in which case we terminate.

An assembly in the list can be either a relative or absolute file path ("SwingInterop.dll" is a relative file path; you may need to modify this if SwingInterop.dll resides in a different location), or, if the assembly is in the Global Assembly Cache (GAC), a fully-qualified assembly name, as in the case of System.Windows.Forms above.

To create the .NET side of the application, start Visual Studio and open the solution file SwingInterop.sln. The SwingInterop solution contains two projects: SampleForms, which contains Form1 and Form2, and SwingInteropDotNetSide, which contains the .NET-side server. You are encouraged to examine the code files and the project settings. Note that the .NET side is configured through the application configuration file app.config. Complete the application by adding to the SwingInteropDotNetSide project a reference to the assembly jnbshare.dll in the JNBridgePro installation folder. Add to your project the file jnbauth_x86.dll or jnbauth_x64.dll or both (depending on whether the application will run as a 32-bit process, a 64-bit process, or either one). When adding the jnbauth dll, make sure that its properties settings include "Copy always." You can now build the solution; it should build without errors. The application is now ready to run.

Running the program

Running the program is simple. First, start the .NET side, either by running the SwingInteropDotNetSide application inside Visual Studio, or by double-clicking on the SwingInteropDotNetSide.exe icon. A console window will open. Next, start the Java side client by

double-clicking on run_swingClient_tcp_no_security.bat. This will run the demo using TCP/binary communications, but without the additional security features. The Java-side client program will start (Figure 9). As previously described, clicking on "Open Form1" will bring up Form1 as a modal dialog (Figure 10), and clicking on "Open Form2" will bring up Form2 as a non-modal dialog (Figure 11). In addition, strings entered in the original Java application's text field will be displayed in Form1 when it is brought up, and any string entered into Form1 will be displayed by the Java application when the modal dialog is dismissed (Figure 12).

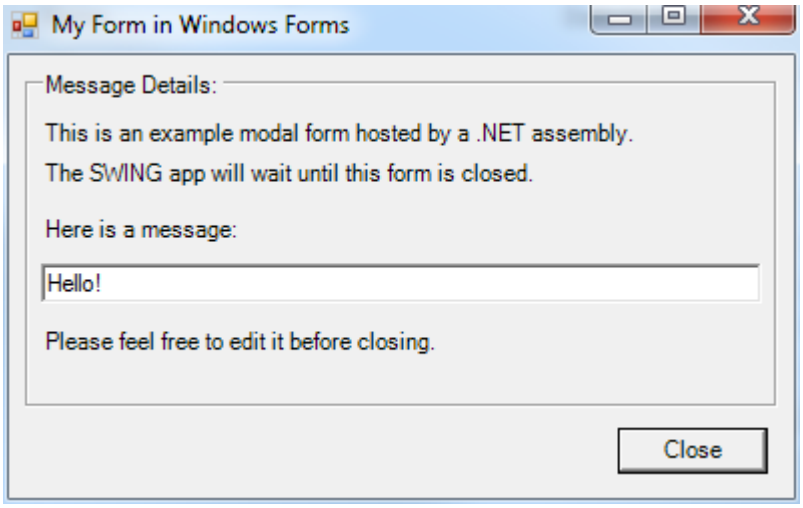

Figure 10. Form1

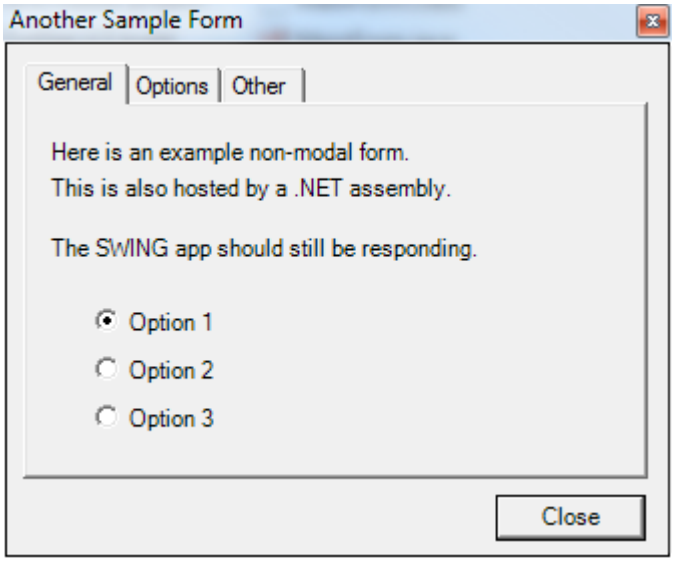

Figure 11. Form2

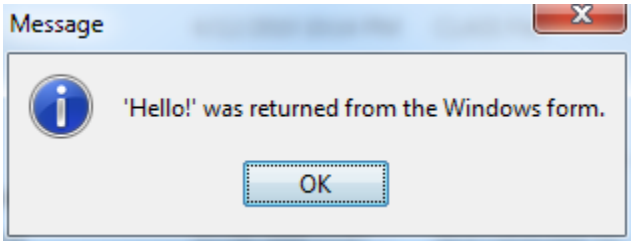

Figure 12. String returned through Form1

Class whitelisting

When using TCP/binary communications, the .NET side can be configured to only allow requests from the Java side that reference specific .NET classes. This prevents the possibility of malicious clients accessing sensitive APIs (which need not even have been proxied). The file App.config in the SwingInteropDotNetSide project contains a variant of <javaToDotNetConfig> that uses class whitelisting. It contains the element useClassWhiteList="true". This is the default value and may be omitted. To turn off class whitelisting, the element must be explicitly set to false.

<javaToDotNetConfig> also contains an element classWhiteListFile=".\classWhiteList.txt". The element above is the path to a text file each of whose lines is a class that can be accessed from the Java side. If the Java side client attempts to access a class not in the whitelist (or one of the short list of classes that is always whitelisted), an exception will be thrown. The supplied whitelist file contains the following classes that are directly accessed from the Java side:

SwingInterop.Form1
SwingInterop.Form2
SwingInterop.JavaWindowEventArgs

The class whitelist can be easily derived by examining the Java side code that calls the proxies. For each proxy class that is called, add that class or interface name to the whitelist.

To use class whitelisting (and SSL) in the demo, uncomment the variant of <javaToDotNetConfig> in App.config that uses the security features, comment out the variant that does not use the security features, and rebuild. Alternatively, directly edit the already-built SwingInteropDotNetSide.exe.config file.

For more information on class whitelisting, see the *Users' Guide*.

Secure communications using SSL

It is possible to configure secure communications between the .NET and Java sides through SSL (secure sockets library). SSL in JNBridgePro provides data encryption, message integrity, and server communications. It is only available when using tcp/binary communications (shared memory is inherently secure). For more information on secure communications, see the *Users' Guide*.

Please note that the following instructions use certificates that we supply. These certificates are for instructional use only; you should NOT use them in production scenarios. For production scenarios, you should supply your own certificates.

To use SSL, first make sure that the example is configured to use tcp/binary communications without SSL (that is, the appropriate useSSL properties are set to false), and that this is working.

Once it is established that the application works with regular tcp/binary communications, we configure for SSL. Use the supplied Java-side properties file jnbcore_tcp_with_security.properties. See the *Users' Guide* for a discussion of the meanings of the various additional properties. You may need to edit the paths in the javaSide.trustStore and javaSide.keyStore properties.

On the .NET side, in the app.config file, comment out the version of <dotNetToJavaConfig> without the security features, and uncomment the version of <dotNetToJavaConfig> with the security features. Note the following elements:

- *useSSL* – this indicates that SSL is being used, and should be set to true
- *serverCertificateLocation* – this is the path to the .NET side's server certificate, and is used to authenticate itself to the server side, and also for encryption. This is a .p12 or .pfx file, and as such should contain both the server's public and private keys.
- *serverCertificatePassword* – this is the password of the server certificate .p12/.pfx file.

On the Java side, we have the following additional properties:

- *javaSide.keyStore* – this is the path to a Java keystore (.jks) file containing the public/private key pair for the Java-side server's certificate (mytestclient).
- *javaSide.keyStorePassword* -- this is the password of the keystore file.
- *javaSide.trustStore* – this is another .jks file containing a list of trusted certificates. You should place the authorized .NET sides' certificates in this folder (dotnetside).
- *javaSide.trustStorePassword* – this is the password of the truststore file.

Since the Java-side server certificate (in this case, myTestClient.cer) is a self-signed certificate, we have to explicitly instruct the .NET side to trust it. To do so, copy the certificate to the .NET-side machine and install it into the certificate store by right-clicking on the .cer file and selecting Install…. In the resulting wizard, choose to install the certificate in either the machine store or the user store. In the next step, when asked where the certificate should be stored, select either "Trusted Root Certification Authorities" or "Third-Party Root Certification Authorities." After that selection, follow all remaining instructions.

In addition, on the .NET-side machine, install the version of the server certificate that contains the public/private key pair (dotnetside.p12) in the Windows certificate store using the instructions above. (You will need to supply the password *changeit* in this case.)

Once you have done all of this, start the .NET side (SwingInteropDotNetSide.exe), then run run_SwingClient_tcp_with_security.bat to start the Java side.

Shared-memory communications

It is also possible to run the .NET side inside the Java process, and to communicate between the two sides using shared memory structures rather than the socket-based binary communications mechanism used in the above example. To change to shared-memory communications, make the following changes to the project:

- Edit the supplied file jnbcore_sharedmem.properties to reflect the paths to jnbjavaentry.dll and SampleForms.dll on your machine.
- Launch runSwingClient_sharedmem.bat to start up the Java side. The .NET side will automatically be started, and the program will run. It's that simple.

Summary

The above example shows how simple it is to integrate Java and .NET code and to run the resulting program. The example above shows how a Java program can pop up .NET windows and gather information from them. It also shows how callbacks can be used in Java-to-.NET projects. Such a strategy greatly simplifies development when an existing Java application with a Swing-based GUI is extended onto the .NET platform..

Creating this program was accomplished in three stages:

- In the first stage, proxies were generated allowing access by Java classes to the .NET classes. The proxies were generated using JNBProxy, a visual tool that allows developers a wide variety of strategies for determining which .NET classes are to be exposed to access by Java (or vice versa).
- In the second stage, the Java jar file containing the proxies was linked into a Java application that used the Java proxies to control .NET Windows Forms. Java classes can access .NET classes transparently, as if the .NET classes were themselves Java classes. Nothing special or additional needs to be done to manage Java-.NET communications or object lifecycles.
- In the third stage, the integrated .NET and Java code is run. All that is required is to start a .NET-side containing the .NET code to be accessed and an additional support module (jnbshare.dll). Once the .NET-side is started, the user simply runs the Java program that will access the Java objects.

By allowing Java and .NET code to interoperate, JNBridgePro helps developers derive full value from their existing Java code, even as they take advantage of Microsoft’s .NET platform.