JNI++ User Guide

Table of Contents

Chapter 1

Whetting Your Appetite

1.1 What is JNI++?

If you've ever written a complex Java Native Interface (JNI) program using the raw procedural interface, then you already understand the tedious, repetitive and error-prone nature of the work. Not only is the work time-consuming (and not particularly interesting), it does not contribute directly to the problem you are trying to solve.

JNI++ is a set of code generating utilities designed to minimize the amount of time required to map between the worlds of C++ and Java. Code can be effortlessly generated to access C++ classes from Java and vice-versa. In addition to the code generators, a core native library is supplied that provides a simplified interface for the *JVM* and the *JNIEnv* for each running thread. This core library also contains wrapper classes for the raw JNI data types and handles the translation between Java and C++ exceptions.

1.2 A Simple Comparison

A common need for the use of JNI is to provide access to legacy C++ code. As a simple comparison, let's assume that we have such a code base and we need to expose some of the functionality of one of the C++ classes. We will first examine the steps required to perform this integration using the raw procedural interface, then using the JNI++ code generating utilities. Just for illustration, we will keep this example short and simple.

1.2.1 The Problem

A subset of the functionality contained in a legacy C++ class needs to be exposed. Here is the class declaration:

```
#ifndef __LegacyImpl_H
#define __LegacyImpl_H

#include <string>

class LegacyImpl
{
private:
        std::string data;

public:

        LegacyImpl()
                : data( "" )
        {
        }

1:
        void loadByID(const std::string& id)
        {
                data = "LegacyImpl ID( " + id + " )";
        }


        std::string getData()
        {
                return data;
        }
};
```

```
#endif
```

Notes:

1. For simplicity, we will just initialize the "data" member with the identifier passed in. A more realistic scenario would involve initialization from a legacy database, given the identifier.

Code Snip 1 - 1: Legacy Code

While this is admittedly a contrived and simple example, it is sufficient to illustrate the differences between the raw solution and that using JNI++. We must be able to access this legacy code from a Java "proxy" class. The "proxy" class must exhibit the following behavior:

1. Expose a default constructor that creates a LegacyImpl "peer" instance in the native code and associates it with the new Java "proxy" instance.
2. Expose a method to initialize the new instance given a `String` identifier. In our example, we will just perform a simple initialization based on the ID passed in. We must also throw a `java.lang.IllegalArgumentException` if a `null` "id" parameter is passed.
3. Expose a "getter" method to retrieve the data associated with the "peer" LegacyImpl instance.
4. Ensure that the "peer" LegacyImpl instance is destroyed when the finalize() method of the Java "proxy" class is called.

Okay, this all sounds pretty simple, right? Let's get started....

1.2.2 The Traditional Solution

1.2.2.1 Step 1: Create the Java Class

The first step toward solving the problem is to create the Java class that contains the `native` declarations that satisfy the requirements. Here is the code for the Java "proxy" class:

```
package rawExample;

public class Simple
{
        static
        {
                System.loadLibrary( "Simple" );
        }
1:
        private long handle = 0;

2:
        protected native void init();
        protected native void uninit();

3:
        public native void loadByID(String id);
        public native String getLegacyData();

4:
        public Simple()
        {
                init();
        }
```

```
5:
        protected void finalize()
        {
                uninit();
        }

6:
        public static void main(String[] args)
        {
                try
                {
                        Simple s1 = new Simple();
                        Simple s2 = new Simple();
                        s1.loadByID( "s1ID" );
                        s2.loadByID( "s2ID" );
                        System.out.println( "s1.getLegacyData() == \"" + s1.getLegacyData() + "\"" );
                        System.out.println( "s2.getLegacyData() == \"" + s2.getLegacyData() + "\"" );
                        s1.loadByID( null );
                }
                catch(Exception ex)
                {
                        System.out.println( "Caught exception:" );
                        ex.printStackTrace();
                }
        }

    }
```

Notes:

1. This field will be utilized to maintain the association of this "proxy" class with its native "peer".
2. The `init()` and `uninit()` methods are called by the constructor and the `finalize()` method, respectively. They will maintain the lifetime of the native "peer" class. More on this later.
3. These two methods provide access to the legacy functionality we need to expose.
4. The default constructor calls the native `init()` method to create and initialize the associated C++ "peer" class. More details later.
5. The `finalize()` method calls the native `uninit()` method to ensure the associated C++ "peer" class is released.
6. This `main()` method will be utilized to test the finished product. The entire body is wrapped in a `try ... catch` block, and an exception should be thrown from the native code on the last call to `loadByID()` with the `null` parameter.

Code Snip 1 - 2: Java Proxy Class

As you can see, the Java "proxy" class is relatively straightforward. The `native` methods are declared and a simple `main()` method has been provided for testing.

1.2.2.2 Step 2: Compile the Java Class

Now that we have defined our Java "proxy" class, the next step is to compile the source. This involves invoking the Java compiler in the target environment, and for now we will ignore the details. The interaction looks something like this:

```
[phil@gatekeeper]$ javac -d . Simple.java
[phil@gatekeeper]$
```

Output Sample 1 - 1: Compiling the Java Proxy Class

We interpret the lack of any interesting results as a successful compilation and continue on to the third step.

1.2.2.3 Step 3: Create the JNI Header File

Now that we have a compiled Java class with the `native` methods defined, we now need to create the native implementation. The first step in all of this is to create method decalarations such that they can be located by the Java Virtual Machine (JVM) at runtime. The Java Development Kit (JDK) includes a tool that will generate a header file with the appropriate method signatures. The `javah` executable utility accepts one or more Java classes as input and generates a header file for each. Here is an example of the interaction:

```
[phil@gatekeeper]$ javah -classpath . -d . rawExample.Simple
[phil@gatekeeper]$
```

Output Sample 1 - 2: Generating the Native Implementation Header File

Again, we accept the lack of interesting output as a successful run. This command results in the creation of a header file, `demo_chapters_ichi_rawExample_Simple.h`, whose contents are below:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class demo_chapters_ichi_rawExample_Simple */

#ifndef _Included_demo_chapters_ichi_rawExample_Simple
#define _Included_demo_chapters_ichi_rawExample_Simple
#ifdef __cplusplus
extern "C" {
#endif
1:

/*
 * Class:     demo_chapters_ichi_rawExample_Simple
 * Method:    init
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_demo_chapters_ichi_rawExample_Simple_init
  (JNIEnv *, jobject);

/*
 * Class:     demo_chapters_ichi_rawExample_Simple
 * Method:    uninit
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_demo_chapters_ichi_rawExample_Simple_uninit
  (JNIEnv *, jobject);

/*
 * Class:     demo_chapters_ichi_rawExample_Simple
 * Method:    loadByID
 * Signature: (Ljava/lang/String;)V
 */
JNIEXPORT void JNICALL Java_demo_chapters_ichi_rawExample_Simple_loadByID
  (JNIEnv *, jobject, jstring);

/*
 * Class:     demo_chapters_ichi_rawExample_Simple
 * Method:    getLegacyData
 * Signature: ()Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_demo_chapters_ichi_rawExample_Simple_getLegacyData
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Notes:

1. The *javah* utility has generated declarations for each of the native methods declared in our Java "proxy" class.

Code Snip 1 - 3: Generated Header File

As you can see, the `javah` utility has performed the work of generating method declarations for the `native` methods declared in our Java "proxy" class. This is helpful, but we have a number of steps remaining before we are finished. We must create the implementation file, fill in the method implementations and build the result.

1.2.2.4 Step 4: Create the JNI Implementation File

The `javah` utility relieves us from understanding how our methods must be named in order to be located by the JVM. Beyond that, however, it does nothing for us. We must still create the implementation file and provide all of the detailed code to map between the two worlds. This next code section supplies an implementation for each of the given `native` methods:

```
   #include "rawExample_Simple.h"
   #include "../LegacyImpl.h"
   #include <map>

   /*
    * Class:     demo_chapters_ichi_rawExample_Simple
    * Method:    init
    * Signature: ()V
    */
   JNIEXPORT void JNICALL Java_demo_chapters_ichi_rawExample_Simple_init(JNIEnv* env, jobject obj)
   {
1:
          jclass cls = env->GetObjectClass( obj );
          jfieldID fid = env->GetFieldID( cls, "handle", "J" );
          LegacyImpl* peer = new LegacyImpl;
          env->SetLongField( obj, fid, reinterpret_cast<jlong>(peer) );

   }

   /*
    * Class:     demo_chapters_ichi_rawExample_Simple
    * Method:    uninit
    * Signature: ()V
    */
   JNIEXPORT void JNICALL Java_demo_chapters_ichi_rawExample_Simple_uninit(JNIEnv *env, jobject obj)
   {
          jclass cls = env->GetObjectClass( obj );
          jfieldID fid = env->GetFieldID( cls, "handle", "J" );
          jlong handle = env->GetLongField( obj, fid );

2:
          LegacyImpl* peer = reinterpret_cast<LegacyImpl*>( handle );
          delete peer;

   }

   /*
    * Class:     demo_chapters_ichi_rawExample_Simple
    * Method:    loadByID
    * Signature: (Ljava/lang/String;)V
    */
   JNIEXPORT void JNICALL Java_demo_chapters_ichi_rawExample_Simple_loadByID(JNIEnv* env, jobject obj,
   jstring str)
   {
3:
          if ( str == NULL )
          {
                 jclass cls = env->FindClass( "java/lang/IllegalArgumentException" );
                 env->ThrowNew( cls, "id parameter cannot be NULL" );
                 return;
          }

          const char* id = env->GetStringUTFChars( str, NULL );
          jclass cls = env->GetObjectClass( obj );
          jfieldID fid = env->GetFieldID( cls, "handle", "J" );
          jlong handle = env->GetLongField( obj, fid );
```

```
4:
        LegacyImpl* peer = reinterpret_cast<LegacyImpl*>( handle );
        peer->loadByID( id );
        env->ReleaseStringUTFChars( str, id );

    }

    /*
     * Class:     demo_chapters_ichi_rawExample_Simple
     * Method:    getLegacyData
     * Signature: ()Ljava/lang/String;
     */
    JNIEXPORT jstring JNICALL Java_demo_chapters_ichi_rawExample_Simple_getLegacyData(JNIEnv* env, job-
    ject obj)
    {
        jclass cls = env->GetObjectClass( obj );
        jfieldID fid = env->GetFieldID( cls, "handle", "J" );
        jlong handle = env->GetLongField( obj, fid );


5:
        LegacyImpl* peer = reinterpret_cast<LegacyImpl*>( handle );
        return env->NewStringUTF( peer->getData().c_str() );

    }
```

Notes:

1. The `init()` method is called from the constructor and is responsible for creating the new legacy implementation and associating it with the new instance. Here we utilize the `handle` field that we defined earlier in the Java "proxy" class to hold a pointer to the legacy class instance.
2. This method is called by the `finalize()` method of our Java "proxy" class. After retrieving the value of the `handle` field of the given Java instance, we cast this into a pointer and `delete` the result. This controls the lifetime of the legacy class instance.
3. We first check to see if an attempt has been made to pass in a `null` "id" parameter. If so, we flag a `java.lang.IllegalArgumentException` to the JVM with the message "id parameter cannot be NULL".
4. After unpacking the `jstring` parameter and finding the associated C++ "peer" class instance, we delegate the `loadByID()` call to that instance.
5. Our `getLegacyData()` method simply delegates the call to the `getData()` method of our legacy C++ class instance after retrieving the `handle` value, which is a pointer to the associated instance.

Code Snip 1 - 4: Native Implementation File

1.2.2.5 Step 5: Create a Suitable Makefile

While this step can be viewed as optional for a small project such as this, realistically it is not. Most projects involve a more complicated build process that necessitates the use of one or more makefiles. The text of the makefile is below.

```
INCLUDES= -I. -I$(JAVA_HOME)/include -I$(JAVA_HOME)/include/$(OS)
TARGETTYPE= shlib
TARGETNAME= libSimple

ifeq ($(TARGETTYPE), shlib)
DBGCPPFLAGS= -O0 -g -fPIC -c -Wall $(INCLUDES)
RELCPPFLAGS= -O0 -fPIC -c -Wall $(INCLUDES)
else
DBGCPPFLAGS= -O0 -g -c -Wall $(INCLUDES)
RELCPPFLAGS= -O0 -c -Wall $(INCLUDES)
endif

LINKFLAGS=
EXT=
ifeq ($(TARGETTYPE), shlib)
ifeq ($(OS), linux)
LINKFLAGS=-shared
EXT=.so
else
SHLIBCMD=-G
EXT=.so
endif
```

```
endif

%_d.o: %.cpp
                        g++ $(DBGCPPFLAGS) $< -o $@

%.o: %.cpp
                        g++ $(RELCPPFLAGS) $< -o $@

SRCS=  rawExample_Simple.cpp

DBGOBJS=$(patsubst %.cpp, %_d.o, $(SRCS))
RELOBJS=$(patsubst %.cpp, %.o, $(SRCS))

all:         Debug Release

Debug: dirs $(DBGOBJS)
                        g++ $(LINKFLAGS) -o Debug/$(TARGETNAME)_d$(EXT) $(DBGOBJS)

Release: dirs $(RELOBJS)
                        g++ $(LINKFLAGS) -o Release/$(TARGETNAME)$(EXT) $(RELOBJS)

dirs: $(dummy)
                @mkdir -p Debug
                @mkdir -p Release

clean: $(dummy)
                        @rm -rf Debug
                        @rm -rf Release

rebuild: clean Debug Release
```

Code Snip 1 - 5: GNU Makefile for Raw JNI Example

1.2.2.6 Step 6: Build the Native Implementation Library

The next (and final) step is to create the native shared library utilizing our implementation and makefile from the previous steps. Glossing over the details, the interaction looks something like this:

```
[phil@gatekeeper]$ gmake Release
g++ -O0 -fPIC -c -Wall -I. -I/usr/java/jdk1.3/include -I/usr/java/jdk1.3/include/linux
rawExample_Simple.cpp -o rawExample_Simple.o
g++ -shared -o Release/libSimple.so  rawExample_Simple.o
```

Output Sample 1 - 3: Building the Native Implementation Library

Okay, no errors in the build -- now we can finally run and test our solution.

1.2.2.7 Step 7: Run the Completed Application

Again, glossing over a few details (like ensuring the `LD_LIBRARY_PATH` is set appropriately), the interaction (hopefully) resembles the following:

```
[phil@gatekeeper]$ java -cp . demo.chapters.ichi.rawExample.Simple
s1.getLegacyData() == "LegacyImpl ID( s1ID )"
s2.getLegacyData() == "LegacyImpl ID( s2ID )"
Caught exception:
java.lang.IllegalArgumentException: id parameter cannot be NULL
        at demo.chapters.ichi.rawExample.Simple.loadByID(Native Method)
        at demo.chapters.ichi.rawExample.Simple.main(Simple.java:37)
[phil@gatekeeper]$
```

Output Sample 1 - 4: Running the Completed Application

Looks like it worked! As you can see, there is considerable overhead involved in maintaining the relationship between each Java instance and its "peer" C++ legacy class instance. Although this may not seem like much code, imagine integrating tens or hundreds of legacy classes in this fashion. You could expect to spend a considerable amount of time coding and testing the integration alone. This example is also very trivial. It does not make use of more complex data types or callbacks into the Java code, both of which increase complexity and, as a result, the time required to code, test and debug.

1.2.3 Solving the Same Problem the Easy Way

Now that we've seen the work required to implement this solution using the raw procedural `JNI`, let's look at how JNI++ can ease the burden of developing the code to integrate Java and C++.

1.2.3.1 Step 1: Define the Interface

The first step toward the JNI++ solution is to define an interface in Java that describes the services required of the native code. This simple interface defines the contract:

```
   package jnippExample;

   public interface Simple
   {
1:
          public void loadByID(String id);
          public String getData();

   }
```

Notes:

1. These two methods describe the services to be exposed from the legacy native code library.

Code Snip 1 - 6: Interface Definition

1.2.3.2 Step 2: Create the Project File

The JNI++ code generators utilize XML project files to guide the process, and these project files can be either created manually or with the help of the supplied GUI. Once the project file has been created, the project file can be saved and later fed to the command-line code generation interface or the code generators can be invoked directly from the GUI.

The following figure shows the JNI++ GUI with no project settings.

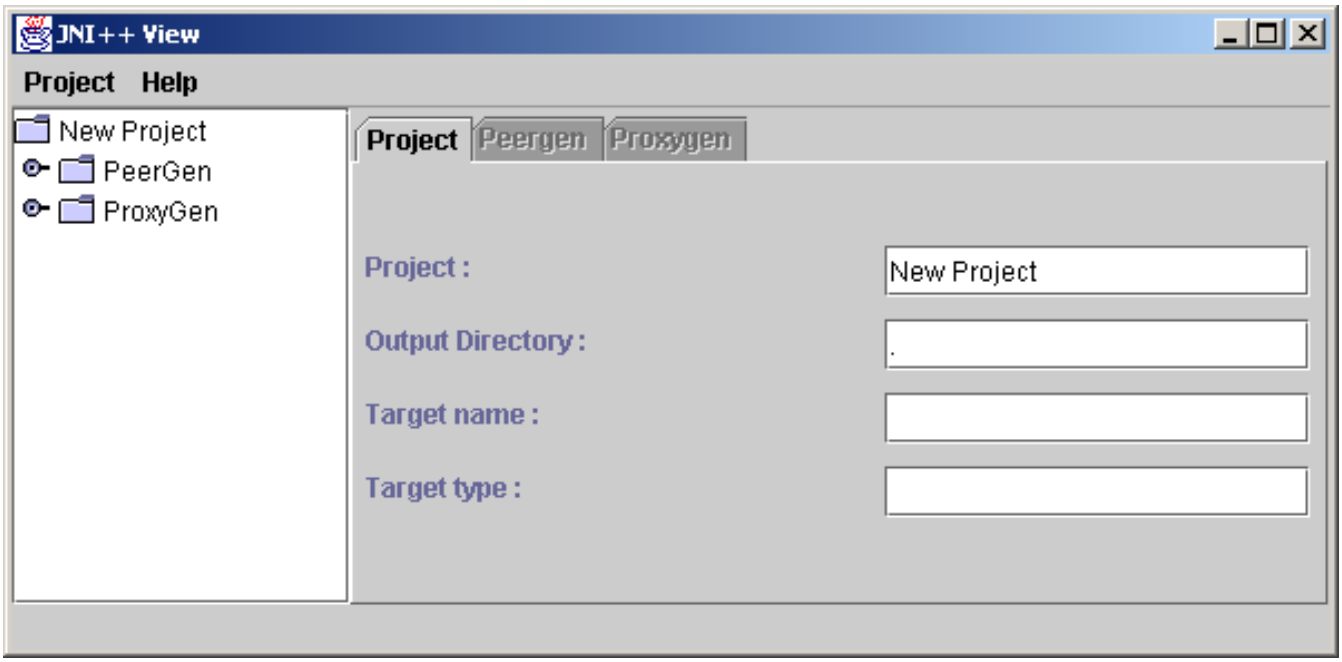

Figure 1 - 1: JNI++ GUI (New Project)

The next two figures illustrate the process of adding a class to the project. In this example, we need to generate a C++ "peer" class that can be accessed from `Java` and we also need a C++ "proxy" to automate the `java.lang.IllegalArgumentException` that we may throw from the C++ "peer" class.

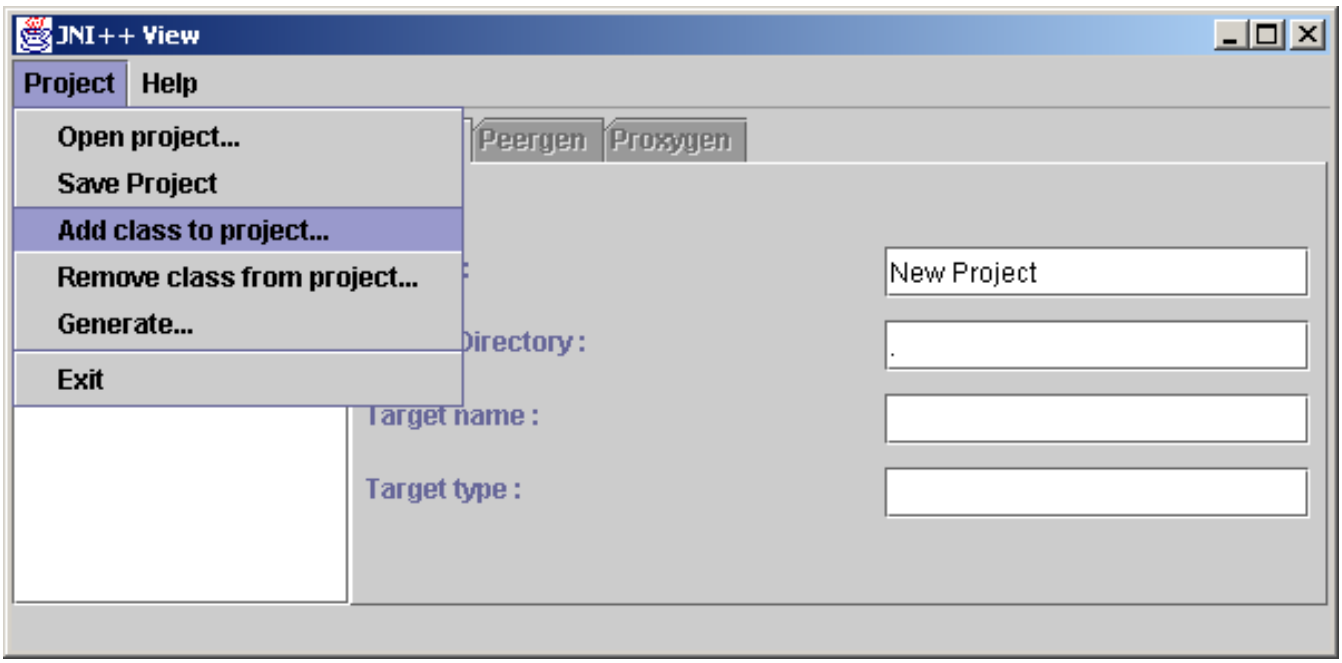

Figure 1 - 2: Adding Input Classes to the Project

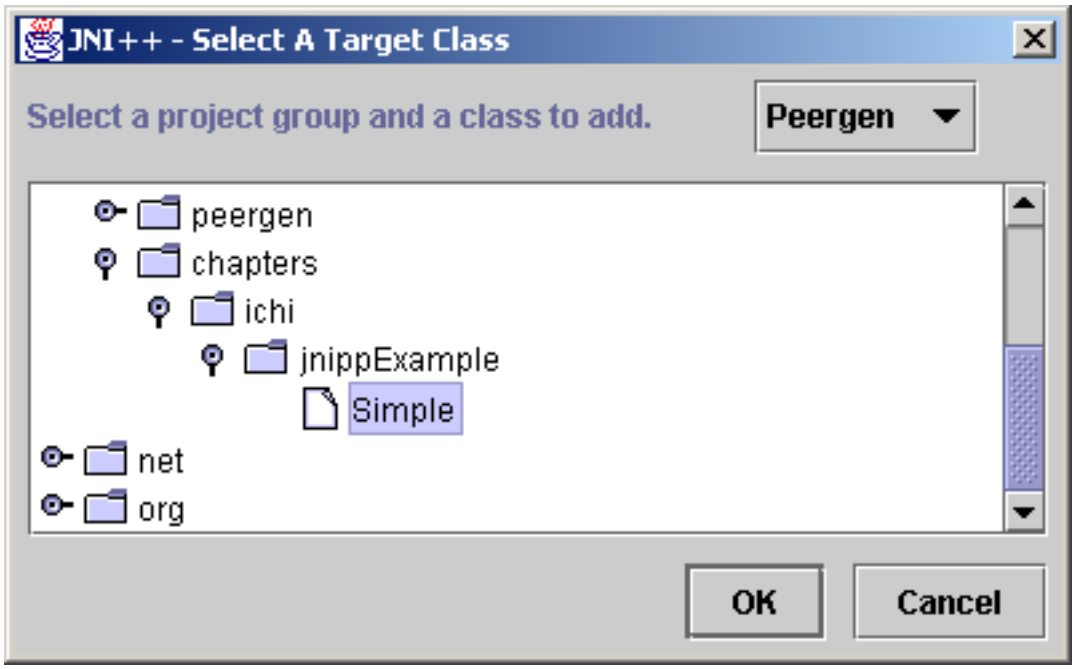

Figure 1 - 3: Adding the Input Class for Peer Generation

The next figure shows a view of the finished project from which we can generate the requisite code.

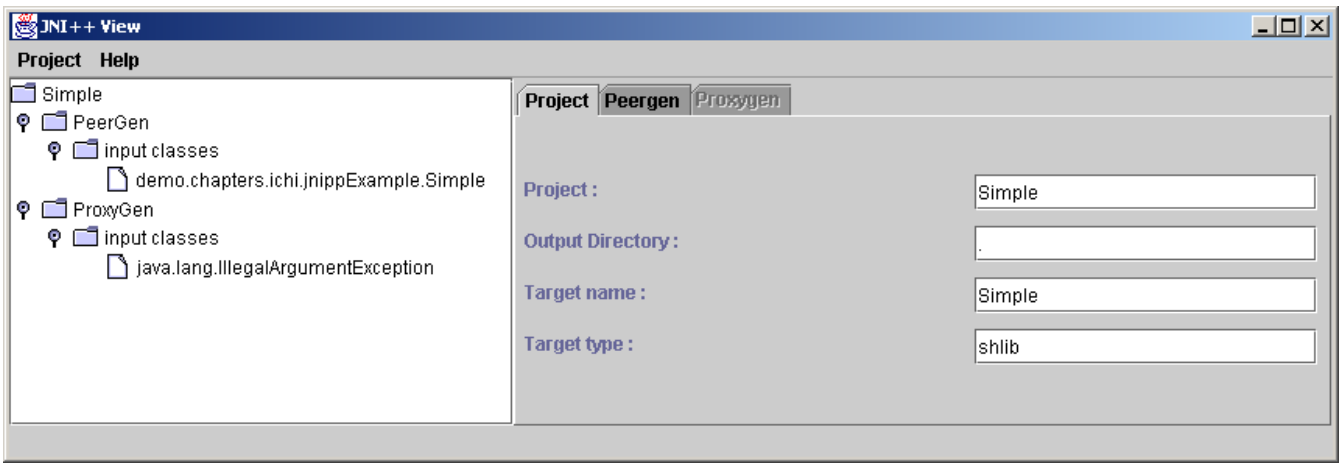

Figure 1 - 4: The Finished Project

Now that we have all of the project settings complete, we can generate the code directly from the GUI. This is illustrated in the next two figures.

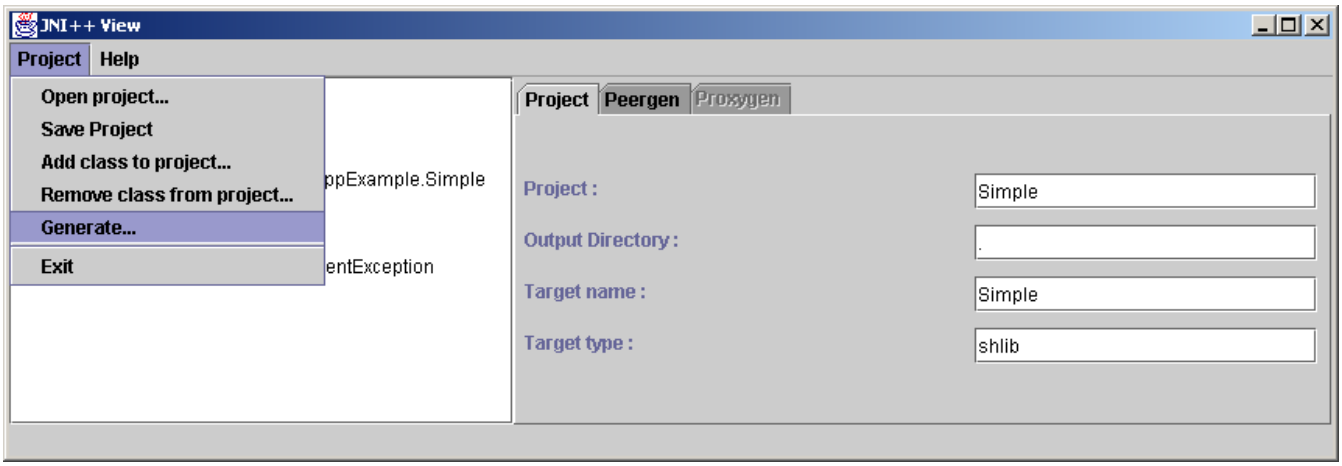

Figure 1 - 5: Invoking the Code Generators from the GUI

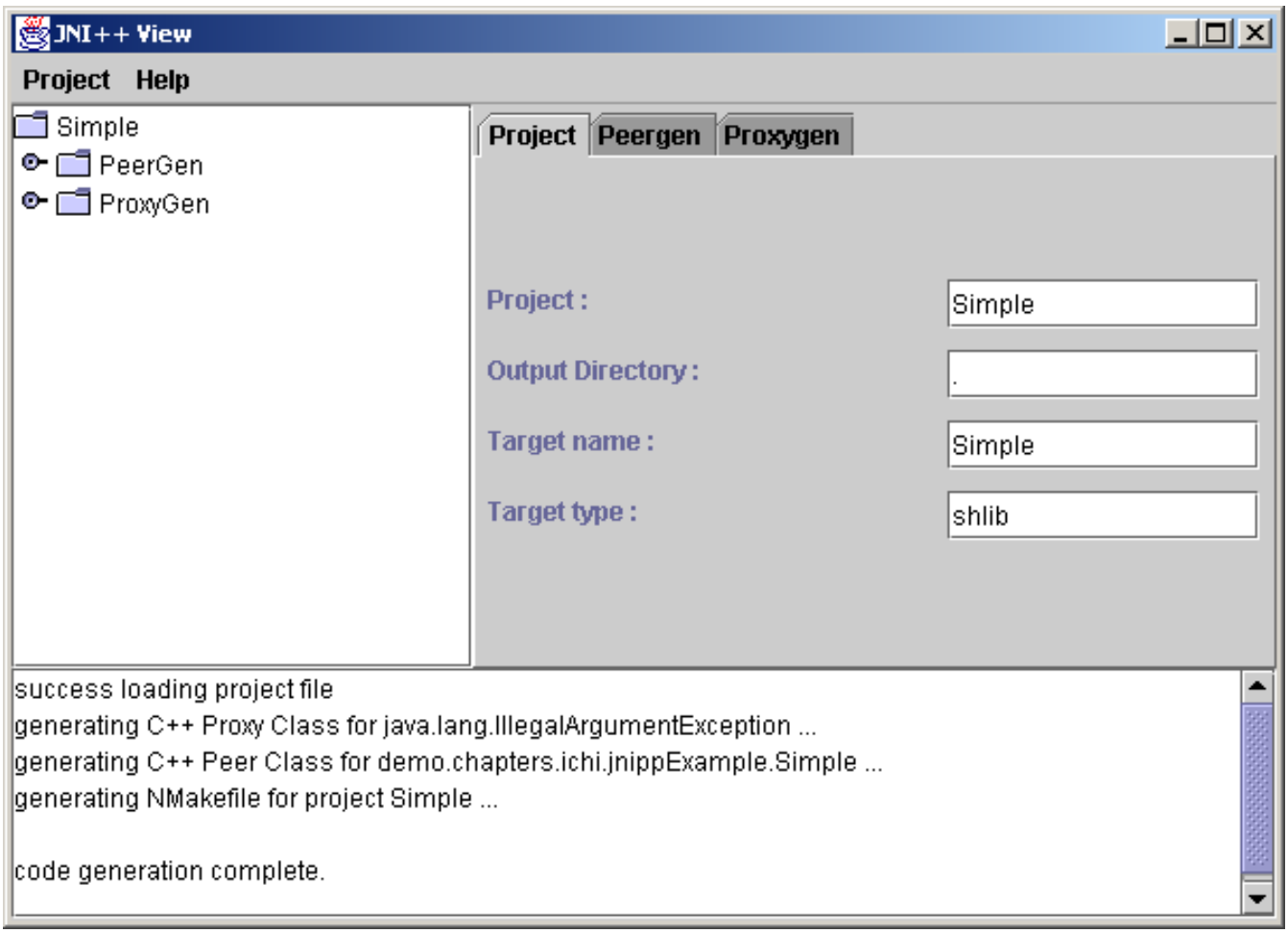

Figure 1 - 6: Code Generation Results

Invoking the code generators results in the production of a C++ "peer" class with its associated mapping code, a C++ "proxy" class for the `java.lang.IllegalArgumentException` and a makefile that can be used to build the target shared library.

1.2.3.3 Step 3: Fill in the Implementation

The JNI++ code generator has created all of the requisite mapping code and C++ "peer" class for us (as well as the Java "proxy" class). All we must do now is fill in the implementation of the generated methods on the C++ "peer" class. Here is the generated header file with modifications:

```
#ifndef __demo_chapters_ichi_jnippExample_SimplePeer_H
#define __demo_chapters_ichi_jnippExample_SimplePeer_H


#include <jni.h>

1:

#include "../../../../../LegacyImpl.h"


namespace demo
{
        namespace chapters
        {
                namespace ichi
                {
                        namespace jnippExample
                        {
                                class SimplePeer
                                {
                                private:

2:

                                        LegacyImpl delegate;


                                public:
                                        SimplePeer();

                                        // methods
                                        jstring getData(JNIEnv*, jobject);
                                        void loadByID(JNIEnv*, jobject, jstring);

                                };
                        };
                };
        };
};
#endif
```

Notes:

1. The `LegacyImpl` header file is included to expose the desired legacy functionality.
2. This `LegacyImpl` instance will serve as the delegate for calls from the `Java` client.

Code Snip 1 - 7: Generated Header File

And here is the generated implementation file with modifications:

```
#include "SimplePeer.h"

1:

#include "java/lang/IllegalArgumentExceptionProxy.h"
#include "net/sourceforge/jnipp/JStringHelper.h"


using namespace demo::chapters::ichi::jnippExample;
using namespace java::lang;
using namespace net::sourceforge::jnipp;

SimplePeer::SimplePeer()
{
}

// methods
jstring SimplePeer::getData(JNIEnv* env, jobject obj)
```

```
{
        // TODO: Fill in your implementation here

2:
        return JStringHelper( delegate.getData().c_str() );

}
void SimplePeer::loadByID(JNIEnv* env, jobject obj, jstring p0)
{
        // TODO: Fill in your implementation here

3:
        if ( p0 == NULL )
        {
                throw static_cast<jthrowable>(IllegalArgumentExceptionProxy( JStringHelper( "id parameter cannot be NULL" ) ) );
        }

        delegate.loadByID( static_cast<const char*>(JStringHelper( p0 )) );
}
```

Notes:

1. The first `include` provides access to the C++ "proxy" class we generated for the `java.lang.IllegalArgumentException` class. The next `include` is for the `JStringHelper` class, one of the JNI++ "helper" classes that simplify working with JNI string types.
2. Here we simply delegate to the contained `LegacyImpl` instance to retrieve the data. Note the use of the `JNIStringHelper` class to simplify the creation of the `jstring` return type.
3. One of the requirements was to throw a `java.lang.IllegalArgumentException` back to the `Java` client if a `null` parameter was supplied. Here we illustrate the use of the generated C++ "proxy" class for this purpose. Note the use of natural C++ constructs to achieve this -- we simply throw the C++ exception "proxy" and it is propagated to the Java client automatically.

Code Snip 1 - 8: Generated Implementation File

1.2.3.4 Step 4: Build the Native Implementation Library

This is a fairly uneventful step consisting of invoking the *make* utility with the generated makefile as input. This will result in the target shared library that can then be utilized to run and test the application.

```
[phil@gatekeeper]$ gmake
[phil@gatekeeper]$
```

Output Sample 1 - 5: Building the Native Implementation Library

1.2.3.5 Step 5: Run the Completed Application

Now that everything is built, we can test the integration. We provide the generated `Java` "proxy" class with the following `main()` method and compile it to the `CLASSPATH`:

```
package demo.chapters.ichi.jnippExample;

import demo.chapters.ichi.jnippExample.Simple;

public class SimpleProxy
        implements Simple
{
        private long peerPtr = 0;

        private static native void init();
        private native void releasePeer();
```

```
        protected void finalize()
                throws Throwable
        {
                releasePeer();
        }

        // methods
        public native String getData();
        public native void loadByID(String p0);


        static
        {
                System.loadLibrary( "Simple" );
                init();
        }

1:

        public static void main(String[] args)
        {
                try
                {
                        SimpleProxy s1 = new SimpleProxy();
                        SimpleProxy s2 = new SimpleProxy();
                        s1.loadByID( "s1ID" );
                        s2.loadByID( "s2ID" );
                        System.out.println( "s1.getData() == \"" + s1.getData() + "\"" );
                        System.out.println( "s2.getData() == \"" + s2.getData() + "\"" );
                        s1.loadByID( null );
                }
                catch(Exception ex)
                {
                        System.out.println( "Caught exception:" );
                        ex.printStackTrace();
                }

        }
}
```

Notes:

1. This `main()` method has been added to the generated `Java` "proxy" class to test the completed application.

Code Snip 1 - 9: Source for Test Driver

After compiling the `Test` class, we run it to produce the following results (again, glossing over the details): This chapter will introduce you to the JNI++ toolset.

```
[phil@gatekeeper]$ java demo.chapters.ichi.jnippExample.SimpleProxy
s1.getData() == "LegacyImpl ID( s1ID )"
s2.getData() == "LegacyImpl ID( s2ID )"
Caught exception:
java.lang.IllegalArgumentException: id parameter cannot be NULL
        at demo.chapters.ichi.jnippExample.SimpleProxy.loadByID(Native Method)
        at demo.chapters.ichi.jnippExample.SimpleProxy.main(SimpleProxy.java:40)
```

Output Sample 1 - 6: Running the Completed Application

As you can see, although the number of steps required to solve the problem is about the same, the complexity of the JNI++ solution is much lower. The code that maps the method calls and field accesses from the generated C++ "proxy" classes to their Java "peer" classes, and from generated Java "proxy" classes to their generated C++ "peer" classes is generated for you. All that is left is to fill in your implemenation for any C++ "peer" classes. The generated C++ "proxy" classes can be utilized as if they were implemented in C++, although behind the scenes the method calls are actally delegated to the corresponding Java "peer" class on the other side of the JNI.

Most of the code to interface Java and C++ can be generated for you. This, in combination with the JNI++ "helper" classes contained in the core runtime library, is a powerful tool that can accelerate your JNI programming productivity. And although much of your work can be automated, there are no restrictions on the use of raw

JNI programming techniques alongside the generated and supplied code. If you feel the need to access raw JNI functionality, it is fully exposed and available for your use.

Chapter 2

Setting Up

2.1 Tested Environments

This release of JNI++ has been successfully tested only with the Sun JDK version 1.3.1 for both Win32 and RedHat Linux 7.1. The one problem encountered with the JDK under RedHat Linux occurs when an exception is thrown within C++ code in a native implementation library. The JVM will abort even though the exception is caught within the C++ code and *never* propagated to the JVM. The workaround is to invoke the JVM with the "-classic" parameter. More information can be obtained through the following link (you must register): http://developer.java.sun.com/developer/bugParade/bugs/4389172.html. The problem will be addressed in an upcoming release.

The generated C++ source files and makefiles have been successfully tested with Microsoft Visual C++ 6.0 and with GNU g++ version 2.96. Although other environments remain untested, the generated source should build with relative ease using any modern C++ compiler.

The focus of this release has been on the core feature set and documentation, not on supporting the various JDKs. That being said, however, you should encounter few, if any, problems using JNI++ with a JDK other than those tested. As the product matures and the feature set is solidified, more effort will be focused on testing and documenting its use with other JDKs and operating systems. Theoretically, the generated code and supporting libraries should work with any JVM that complies with the Java 2 JNI Specification.

2.2 Installing JNI++

2.2.1 Upgrading from a Previous Release

If you are upgrading from a previous release of JNI++, it is recommended that you remove all files associated with the previous release and re-generate your proxy and peer source. The software has not yet reached a point where it will not experience significant changes between minor releases (it is still *alpha*), and this is one of those releases. In the future (as the software enters *beta*), a more palatable upgrade path will be available.

2.2.2 Requirements

JNI++ utilizes an XML format for projects, and as a result, is dependent upon the *Xerces2* XML parser to operate. This library must be in the `CLASSPATH` when the JVM is invoked. The latest version of this library can be downloaded from the Apache XML Project site, http://xml.apache.org/xerces2-j/index.html. Optionally, the *Ant* build tool can be utilized to simplify your build process and can be downloaded from the Jakarta Project site at http://jakarta.apache.org/ant/index.html.

You will also obviously need a suitable Java Runtime Environment, but if you didn't already know that, then you probably wouldn't be particularly interested in this product anyhow. The same goes for a C++ compiler. Check the beginning of this chapter for compatibility.

2.2.3 Unpacking the Distribution

The JNI++ toolset is distributed as a *zipped* or *gzipped/tarred* archive. Unzipping the distribution results a directory structure similar to the following:

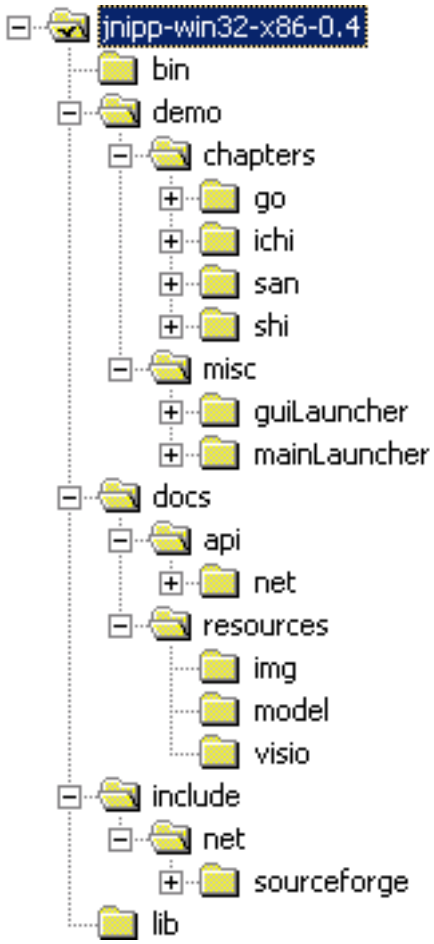

Figure 2 - 1: JNI++ Distribution Directory Structure

The image above shows a few levels deep into the Win32 distribution. The content of these directories is summarized below.

Directory	Contents
bin	This directory is home to the `jnipp` and `jnippGUI` executable images. These are simply executable wrappers that invoke the Java Virtual Machine and then call the `main()` method of the `net.sourceforge.jnipp.main.Main` and `net.sourceforge.jnipp.gui.App` classes, respectively. Not surprisingly, these wrappers were generated using the JNI++ Proxy Generator.
demo	This directory contains all of the sample projects distributed with the release. The build files are written for use with the *Ant* build tool, so you may want to download and configure it before exploring the examples. See the "Requirements" section above for details.
docs	This directory contains all of the user documentation (including the one you are reading) as well as the generated *Javadoc* API documentation (in HTML format).
include	This directory contains all of the header files for the core JNI++ runtime library.
lib	All of the debug and release shared and link libraries for the core JNI++ runtime as well as the single *jnipp.jar* file are located here.

2.2.4 Setting up the Environment

The JNI++ code generators are written in Java, and therefore can be used by adding `jnipp.jar` to your `CLASSPATH` and invoking the command line `java net.sourceforge.jnipp.main.Main`. Beginning with the 0.4 release, executable wrappers have been generated for both the command-line tools and the GUI. These files can be found in the `bin` directory of the distribution (see above). These executables utilize a few environment variables for initialization. The `JNIPP_HOME` environment variable must be set to the base of the JNI++ distribution, such as `C:\jnipp-win32-x86-0.4` or `/usr/local/jnipp-linux-x86-0.4`. For convenience, the `JNIPP_HOME/bin` directory should be added to the `PATH` environment variable. The `JNIPP_HOME/lib` directory should be added to the `PATH` in a Win32 environment and to the `LD_LIBRARY_PATH` under Linux (alternatively you can add it to the list in `/etc/ld.so.conf`). The `JVM_HOME` variable is the last that needs to be set and identifies the JVM that will be loaded. Unfortunately, the process of finding and loading the target JVM is different between the two supported platforms.

In a Win32 environment, the `JVM_HOME` environment variable is used to locate the JVM DLL, and there are a few options for setting its value. It can be simply set to the full path of the target JVM shared library, for example `C:\java\jdk1.3\jre\bin\hotspot\jvm.dll`. Alternatively, the `PATH` environment variable can be set to the directory containing the JVM shared library and then the `JVM_HOME` variable can be simply set to the name of the shared library. In our example, the `PATH` would be updated to include `C:\java\jdk1.3\jre\bin\hotspot` and the `JVM_HOME` would simply be set to "jvm.dll".

Setting up the environment to run under RedHat Linux is a bit more involved, and I haven't quite figured out why it needs to be so different. Much to my disappointment, it was discovered that I could not simply set the `JVM_HOME` variable to the absolute path of the target JVM, such as `/usr/java/jdk1.3/jre/lib/i386/client/libjvm.so`. Next, I tried setting the `LD_LIBRARY_PATH` to include the `/usr/java/jdk1.3/jre/lib/i386/client` directory and managed to get a little further -- now it cannot load `libverify.so`. This file is one of the common JVM shared libraries that lives in the parent directory of the installed JVMs. It was not until I added this parent directory, `/usr/java/jdk1.3/jre/lib/i386` to the `LD_LIBRARY_PATH` that I achieved success. I don't understand why this extra step is required. It seems that the Win32 JVM "knows" that these common libraries are in the parent directory, but the Linux JVM is not quite that smart. I was also, for some mysterious reason, unable to resolve the problem by adding these two directories to the `/etc/ld.so.conf`. This is the only configuration that works for my installation.

Although this will get you started, there are other environment variables that will need to be set if you wish to utilize JNI++ generated makefiles or work through the provided examples. The makefiles generated by JNI++ use the `JAVA_HOME` environment variable to build the include path and should be set to the base of the target JDK, for example `C:\java\jdk1.3`. The generated makefiles are also dependent upon the `JNIPP_HOME` environment variable being set as specified in the previous paragraph. UNIX user must also set the `OS` environment variable such that the `include` directive in the `jni.h` file for operating system specifics can be resolved. As of the 0.4 release, the only supported UNIX flavor is linux, and the `OS` should be set to "linux".

2.3 Deploying JNI++ Applications

2.3.1 Resolving Dependencies

The source code generated from the JNI++ toolset is dependent upon a single shared library, `libJNIPPCore` (distributed as a DLL under Win32 and as a SO or SL under UNIX). This library, in turn, is dependent upon the following (assuming that you are using one of the binary distributions):

Win32:

Library Name	Description
advapi32.dll	This is a standard system DLL that should be available on the system.
kernel32.dll	This is another standard system DLL that should be installed as part of the OS.
msvcrt.dll	This is the C Runtime Library (CRT) deployed with the Microsoft Visual C++ and will need to

Library Name	Description
	be installed if it is not already available.
msvcp60.dll	This library contains support routines for the Standard Template Library (STL) code that is used in the JNI++ core runtime library (libJNIPPCore.dll). This library will need to be installed if it is not already available.

Linux:

Library Name	Description
libstdc++-libc6.2-2.so.3	The standard C++ library.
libdl.so.2	For dynamic loading of shared libraries (the JVM).
libpthread.so.0	Multithreading support.
libm.so.6	Part of the glibc package.
libc.so.6	The C Runtime Library.
/lib/ld-linux.so.2	Support for dynamic loading of shared libraries under Linux.

In general, all of the libraries above, in some version, should already be installed on your Linux system. If not, they are all freely available and easy to locate using your favorite search engine.

If you are attempting to use JNI++ in an environment that does not match those listed above, chances are that you will need to build a distribution. Please see the section titled *Building JNI++* below for more details.

2.3.2 Setting up the Deployment Environment

Deploying a JNI++ application requires copying the shared object and executable files that comprise the application along with the required files (see preceding section) to the target system. Successful deployment also involves ensuring the appropriate environment variables are set such that all of the dependencies can be resolved. Needless to say, a compatible Java Runtime Environment (JRE) also must reside on the target system.

In general, the `CLASSPATH` needs to be updated to include any Java classes, generated or otherwise, that are referenced in your application. If you are deploying an executable image that invokes the JVM, then you may need to set the `JVM_HOME` environment variable, depending on the form of the `JVM::load()` method is used. See the beginning of this chapter for details on setting this variable. If you are deploying C++ peer implementation shared libraries, then these need to be in a place where they can be loaded by the system. In a Win32 environment, this includes the "current working directory" and the directories referenced in the `PATH` environment variable. Under UNIX, this usually includes the directories referenced in the `LD_LIBRARY_PATH` or listed in the `/etc/ld.so.conf` file.

2.4 Building JNI++

Chapter 3

The JNI++ Proxy Generator

3.1 Overview

Each of the two JNI++ code generators is responsible for generating C++ and Java source that enables unidirectional communication. The first of these is the C++ Proxy Generator. This utility is responsible for generating code to enable C++ source to instantiate and use Java classes as if they were implemented in C++. Where Java uses packages, the generated C++ code uses namespaces. Where Java uses primitives and other Java objects as field, parameter and return value types, the generated C++ code uses native JNI types, JNI++ helper classes and generated C++ proxy classes. Depending on the project settings, the use of the generated C++ code has a remarkably similar feel to that of Java. This chapter will get you started with the JNI++ Proxy Generator by stepping through annotated examples that illustrate the impact of the various project settings. The source for all of the examples is located in the `JNIPP_HOME/demo/chapters` directory of the distribution.

The JNI++ Proxy Generator accepts a project file and one or more Java classes and interfaces as input, and generates all of the code you need to conveniently utilize the Java classes as if they were written in C++. It can also optionally generate makefiles for your target platforms, so building the completed JNI++ application is a snap. The figure below illustrates the steps followed for a typical project.

The first step is to create a project file that lists the input Java classes and sets various code generation options. Next, the input Java classes are compiled. This second step is required because the JNI++ code generators do not parse the Java source files, but rather rely on Java introspection to query the various metadata for a given input class. Next, the JNI++ Proxy Generator is invoked to generate C++ proxy classes for the defined inputs. Using either the generated makefiles or some other build environment, the generated C++ proxy classes are compiled and linked with user-written code and the core JNI++ runtime library to produce either an executable image or shared library.

All of the examples in this chapter follow this exact set of steps to build an executable image with simple driver code to exercise the generated proxy classes. You may find it helpful to refer to this diagram when stepping through the examples.

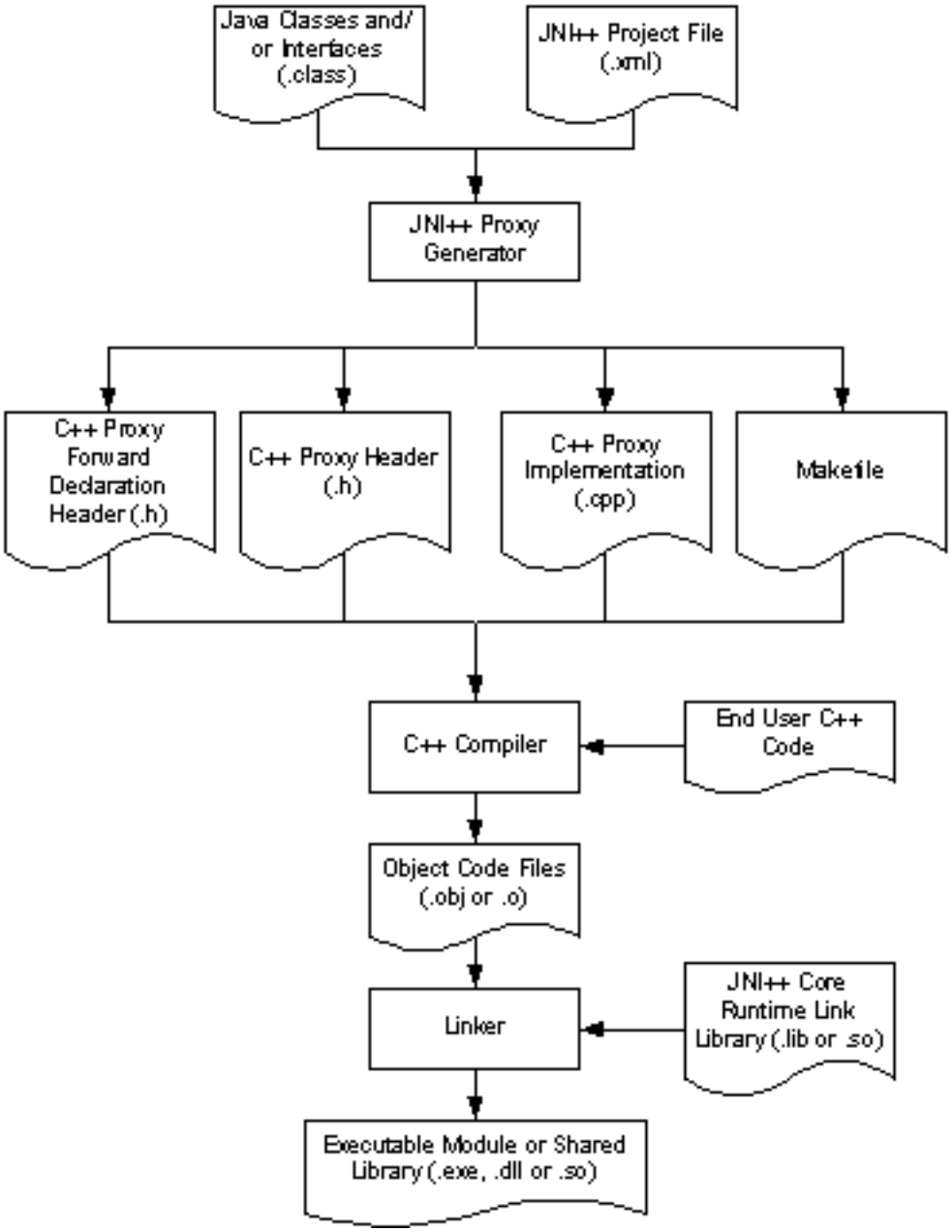

Figure 3 - 1: JNI++ Proxy Generation

The figure below shows a typical runtime execution environment for an application that is using generated C++ proxy classes. The generated C++ proxy classes are used by the client code, which utilizes the simplified JVM interface provided by the JNI++ core runtime library. The core runtime library, in turn, translates calls to the loaded Java virtual machine. Note that although the generated proxy classes provide a simple interface, the developer is always free to bypass either the generated code by calling the routines in the core runtime library directly. If necessary, both the generated code *and* the core runtime library can be bypassed by calling into the loaded JVM directly. The option to bypass the various layers of JNI++ provides the ultimate in both simplicity and flexibility.

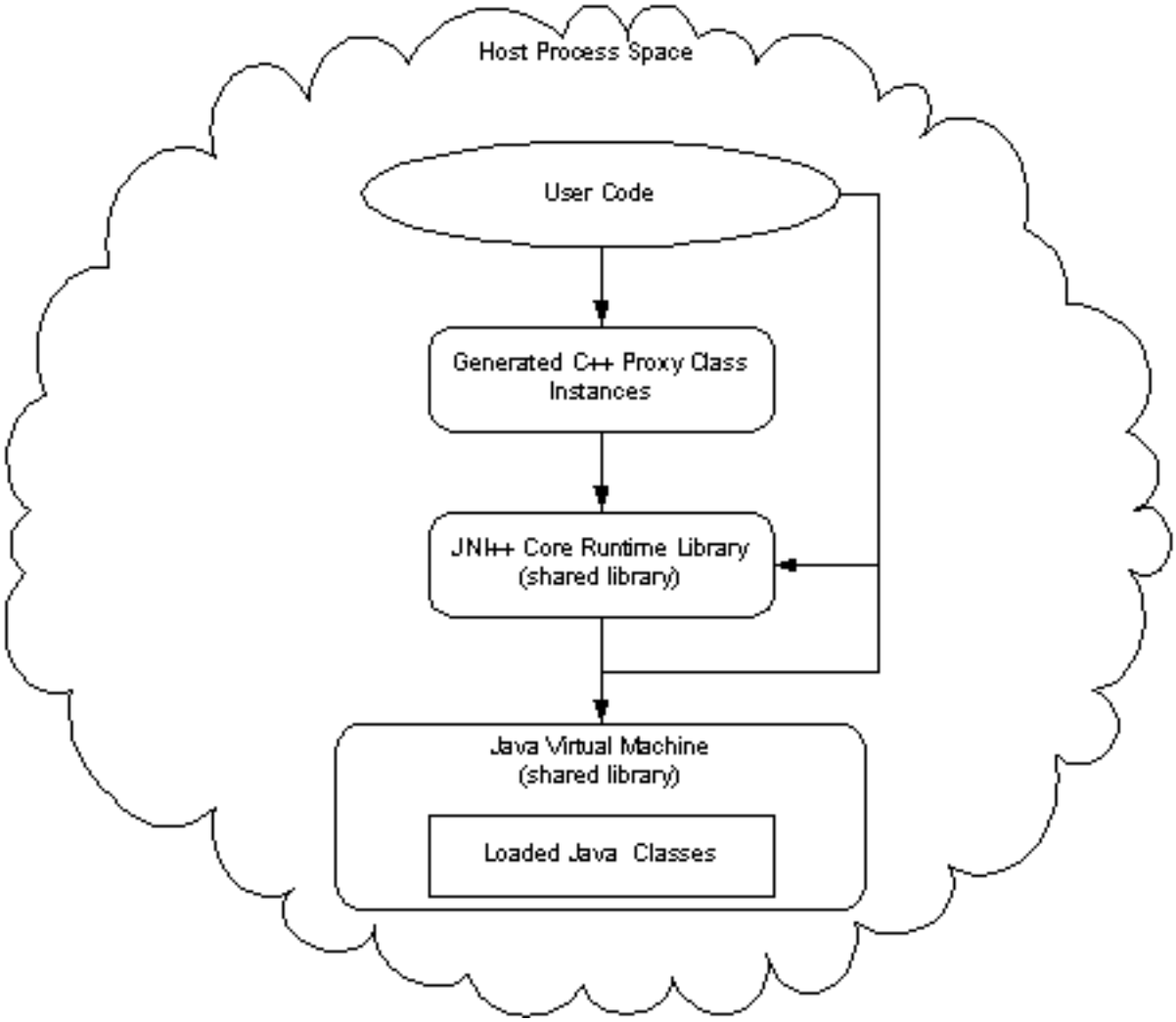

Figure 3 - 2: JNI++ Execution Environment for Proxy Generated Code

3.2 A Very Simple Example

In its simplest form, the JNI++ Proxy Generator can be used to generate a C++ proxy class for a given Java class that uses only raw JNI data types for field, parameter and return types. This first section will illustrate this simple case with an appropriate example. This example will be expanded in the following sections as each new feature is introduced.

3.2.1 The Input Java Class

A C++ proxy class can be generated for *any* Java class, however for simplicity we will define a very simple Java class from which we will base the examples. This class has no value other than for illustrating the features of the code generators. Proxy classes are usually generated for Java classes that already exist.

```
package demo.chapters.san.simple;

public class ProxyDemo
{
        public void printValue(Integer value)
```

```
        {
                System.out.println( "value is " + value );
        }
}
```

Code Snip 3 - 1: The Input Java Class

The JNI++ code generators rely on Java introspection to query the various attributes of the input classes, and as a result the input Java classes must be compiled. This and the following examples assume that the Java classes are compiled into the `%JNIPP_HOME%\build\classes` directory. The following output sample shows the Java class compiled in a Win32 environment.

```
C:\development\jnipp\demo\chapters\san>javac -d %JNIPP_HOME%\build\classes simple\SimpleDemo.java
C:\development\jnipp\demo\chapters\san>
```

Output Sample 3 - 1: Compiling the Java Class

3.2.2 Creating the Project File

The JNI++ code generators rely on an XML project file to provide specifics regarding the code to be generated. The project file can be created with the GUI or with a text editor. The examples in this and the next few chapters will show the text of the project file, beginning with our simple example.

Each project file can contain one or more `proxygen` elements that, in turn, contain one or more `input-class` elements. The attributes of the `proxygen` element specify the settings to be used when generating C++ proxy classes for its `input-class` elements. Most of the attributes have default values so that they need to specified only if the defaults are not desired. The project file for our simple example is shown below with annotations.

```
<?xml version="1.0"?>

1:
<project name="simple" targetType="exe" targetName="simpleDemo">

2:
        <proxygen useRichTypes="false">
                <input-classes>
                        <input-class name="demo.chapters.san.simple.SimpleDemo"/>
                </input-classes>
        </proxygen>

3:
        <nmakefile name="simple.mak"/>
        <gnumakefile name="Makefile"/>

</project>
```

Notes:

1. This `project` element is required and specifies settings that apply to the entire project. Here we specify that the generated files will live in an executable module and that the name of that module will be *simpleDemo*.
2. A project can contain any number of `proxygen` elements. This element directs the code generator to generate C++ proxy classes for the

given input classes. Additionally, we specify that we are not using "rich types" and accept the default settings for all other attributes. The attributes of the `proxygen` element are the topic of this chapter.

3. Each project can generate one makefile for each supported target platform. Here we specify that a makefile should be generated for both the Win32 and Linux platforms with the specified names.

Code Snip 3 - 2: The Project File

3.2.3 Invoking the Code Generator

The JNI++ Proxy and Peer Generators can be invoked from the command line or through the Project Manager GUI. The examples in this guide will show the invocation of the command line tool. Each of the examples in the distribution has an associated Ant build file that can be used to simplify the build process. For this example, however, the steps required to build and run the application manually will be illustrated. Subsequent examples will utilize the Ant build file to build and run.

The `jnipp` executable is invoked with a single command line parameter, *-projectFile* , to specify the project file to use for the code generation settings. The `CLASSPATH` must contain not only the the `xerces.jar` file, but also the classes for which you are generating code. The `JVM_HOME` and `JNIPP_HOME` environment variables must also be set according to the guidelines listed in Chapter 2.

```
C:\development\jnipp\demo\chapters\san\simple>set JNIPP_HOME=c:\development\jnipp\dist

C:\development\jnipp\demo\chapters\san\simple>set JVM_HOME=c:\java\jdk1.3\jre\bin\hotspot\jvm.dll

C:\development\jnipp\demo\chapters\san\simple>set
CLASSPATH=c:\jars\xerces.jar;%JNIPP_HOME%\build\classes

C:\development\jnipp\demo\chapters\san\simple>jnipp -projectFile project.xml
generating C++ Proxy Class for demo.chapters.san.simple.SimpleDemo ...
generating NMakefile for project simple ...
generating GNU Makefile for project simple ...

code generation complete.

C:\development\jnipp\demo\chapters\san\simple>
```

Output Sample 3 - 2: Invoking the Code Generator

3.2.4 A Tour of the Generated Files

So what exactly was produced? This section will walk you through the various files that are generated when the JNI++ Proxy Generator does its thing.

3.2.4.1 The C++ Proxy Headers

At a minimum, a C++ proxy class is generated for each input Java class. Other proxy classes may be generated depending on the project settings and we will see several examples shortly. For the simple case, however, one proxy class is generated for each input Java class.

Each generated C++ proxy class results in two header files and one implementation file, for a total of three source files. The first of the header files simply provides a forward declaration and its intended use is for the `include` directives in the generated header files of other proxy classes. When using the "rich types" feature, it helps prevent cyclic dependencies in the generated proxy header files. It can, however, be used in other source files if needed. We will see examples of the "rich types" feature shortly.

```
#ifndef __demo_chapters_san_simple_SimpleDemoProxyForward_H
#define __demo_chapters_san_simple_SimpleDemoProxyForward_H


namespace demo
{
        namespace chapters
        {
                namespace san
                {
                        namespace simple
                        {
                                class SimpleDemoProxy;
                        }
                }
        }
}

#endif
```

Code Snip 3 - 3: The C++ Proxy Forward Header

The proxy class definition is deferred to the other generated header file shown below. All generated C++ proxy classes have a similar layout, although the content may differ slightly based on the project settings.

```
#ifndef __demo_chapters_san_simple_SimpleDemoProxy_H
#define __demo_chapters_san_simple_SimpleDemoProxy_H


#include <jni.h>
#include <string>


namespace demo
{
        namespace chapters
        {
                namespace san
                {
                        namespace simple
                        {
                                class SimpleDemoProxy
                                {
                                private:
                                        static std::string className;
                                        static jclass objectClass;
                                        jobject peerObject;

                                protected:
                                        SimpleDemoProxy(void* unused);
                                        virtual jobject _getPeerObject() const;

                                public:
                                        static jclass _getObjectClass();
                                        static inline std::string _getClassName()
                                        {
                                                return className;
                                        }

                                        jclass getObjectClass();
1:
                                        operator jobject();

                                        // constructors
2:
                                        SimpleDemoProxy(jobject obj);

3:
                                        SimpleDemoProxy();
```

```
4:                                          virtual ~SimpleDemoProxy();
                                            SimpleDemoProxy& operator=(const SimpleDemoProxy& rhs);

                                            // methods
                                            /*
                                             * void printValue(Integer);
                                             */
5:                                          void printValue(jobject p0);


                                    };
                            }
                    }
            }
    }

    #endif
```

Notes:

1. All generated C++ proxy classes can be cast to a `jobject`, allowing them to be conveniently accepted anywhere a `jobject` parameter is expected.
2. A constructor that accepts a `jobject` parameter is supplied for every generated C++ proxy class, enabling a proxy class instance to be utilized to wrap a `jobject` obtained elsewhere (we will see an example of this shortly).
3. A constructor is generated in the C++ proxy class for each constructor defined in the input Java class. Note that a default constructor is generated even though the input class did not define one. The Java compiler will create a default constructor for a Java class if no other constructors are defined.
4. A destructor and assignment operator are both standard equipment for generated C++ proxy classes.
5. Each method defined in the input Java class results in an associated method in the C++ proxy class. Our input Java class defined only a single method, and here is the C++ proxy analog.

Code Snip 3 - 4: The C++ Proxy Header

3.2.4.2 The C++ Proxy Implementation

```
    #include "net/sourceforge/jnipp/JNIEnvHelper.h"
    #include "SimpleDemoProxy.h"

    using namespace net::sourceforge::jnipp;
    using namespace demo::chapters::san::simple;


    std::string SimpleDemoProxy::className = "demo/chapters/san/simple/SimpleDemo";
    jclass SimpleDemoProxy::objectClass = NULL;

    jclass SimpleDemoProxy::_getObjectClass()
    {
            if ( objectClass == NULL )
                    objectClass = static_cast&lt;jclass&gt;( JNIEnvHelper::NewGlobalRef( JNIEn-
    vHelper::FindClass( className.c_str() ) ) );

            return objectClass;
    }

1:

    SimpleDemoProxy::SimpleDemoProxy(void* unused)
    {
    }

    jobject SimpleDemoProxy::_getPeerObject() const
    {
            return peerObject;
```

```
}

jclass SimpleDemoProxy::getObjectClass()
{
        return _getObjectClass();
}

SimpleDemoProxy::operator jobject()
{
        return _getPeerObject();
}

// constructors
SimpleDemoProxy::SimpleDemoProxy(jobject obj)
{
        peerObject = JNIEnvHelper::NewGlobalRef( obj );
}
```

2:

```
SimpleDemoProxy::SimpleDemoProxy()
{
        jmethodID mid = JNIEnvHelper::GetMethodID( _getObjectClass(), "&lt;init&gt;", "()V" );
        peerObject = JNIEnvHelper::NewGlobalRef( JNIEnvHelper::NewObject( _getObjectClass(), mid ) );
}
```

3:

```
SimpleDemoProxy::~SimpleDemoProxy()
{
        JNIEnvHelper::DeleteGlobalRef( peerObject );
}
```

4:

```
SimpleDemoProxy& SimpleDemoProxy::operator=(const SimpleDemoProxy& rhs)
{
        JNIEnvHelper::DeleteGlobalRef( peerObject );
        peerObject = JNIEnvHelper::NewGlobalRef( rhs.peerObject );
        return *this;
}
```

```
// methods
```

5:

```
void SimpleDemoProxy::printValue(jobject p0)
{
        static jmethodID mid = NULL;
        if ( mid == NULL )
                mid = JNIEnvHelper::GetMethodID( _getObjectClass(), "printValue",
 "(Ljava/lang/Integer;)V" );
        JNIEnvHelper::CallVoidMethod( _getPeerObject(), mid, p0 );
}
```

Notes:

1. This constructor is used by derived class constructors when the `useInheritance` attribute is set. We will see this illustrated shortly.
2. The generated constructors all follow a similar pattern, differing only in the parameters passed to the `NewObject()` method call. The method id is retrieved and the JNI is called to construct the Java class.
3. The generated destructor implementation frees the `peerObject` with a call to `DeleteGlobalRef()`.
4. The assignment operator replaces the `peerObject` with that of the parameter.
5. Here is the implementation of our single method, `printValue`. All of the generated methods have a similar implementation, differing in the method whose id is retrieved, how the method is called and whether any parameters are returned.

Code Snip 3 - 5: The C++ Proxy Implemenation

3.2.4.3 The Generated Makefiles

Looking in the directory from which the code generator was invoked, you will notice the makefiles that were spec-

ified in the project. These makefiles can be utilized to build the specified target, in this case an executable image. Although these makefiles were originally generated by JNI++, they have been modified and checked in because the `Main.cpp` source file must be included in the build (this will be covered in a bit).

```
!IF "$(CFG)" != "Release" && "$(CFG)" != "Debug"
!MESSAGE You must specify a configuration by defining the macro CFG on the command line. For example:
!MESSAGE
!MESSAGE NMAKE /f simple.mak CFG="Debug"
!MESSAGE
!MESSAGE Possible choices for configuration are:
!MESSAGE
!MESSAGE "Release"
!MESSAGE "Debug"
!MESSAGE
!ERROR An invalid configuration is specified.
!ENDIF

!IF "$(OS)" == "Windows_NT"
NULL=
!ELSE
NULL=nul
!ENDIF

1:

TARGETTYPE=CONSOLEAPP


2:

SOURCES= .\demo\chapters\san\simple\SimpleDemoProxy.cpp
SOURCES=$(SOURCES) Main.cpp


OBJS=$(SOURCES:.cpp=.obj)

CPP=cl.exe
INC=/I .\ /I "$(JAVA_HOME)/include" /I "$(JAVA_HOME)/include/win32" /I "$(JNIPP_HOME)/include"
CPPFLAGS=/nologo /GX /W3 /FD /D "WIN32" /D "_WINDOWS" /D "_MBCS" $(INC) /c
LINK=link.exe
LINKFLAGS=/nologo /machine:IX86 /libpath:"$(JNIPP_HOME)/lib"

!IF "$(TARGETTYPE)" == "DLL"
CPPFLAGS=$(CPPFLAGS) /D "_USRDLL"
EXT="dll"
LINKFLAGS=$(LINKFLAGS) /incremental:no /dll
!ELSEIF "$(TARGETTYPE)" == "CONSOLEAPP"
EXT="exe"
LINKFLAGS=$(LINKFLAGS) /subsystem:console /incremental:no
!ENDIF

!IF "$(CFG)" == "Debug"

CPPFLAGS=$(CPPFLAGS) /MDd /Zi /Od /D "_DEBUG"
OUTDIR=Debug
LINKFLAGS=$(LINKFLAGS) /debug
LINKOBJS=$(OBJS:.obj=_d.obj)
LINKLIBS=libJNIPPCore_d.lib

TARGETNAME=simpleDemo_d

"$(OUTDIR)":
        if not exist "$(OUTDIR)\$(NULL)" mkdir "$(OUTDIR)"

.cpp.obj:
        $(CPP) @<< $(CPPFLAGS) /Fo$(<:.cpp=_d.obj) $<
<<

!ELSE

CPPFLAGS=$(CPPFLAGS) /MD /O2 /D "NDEBUG"
OUTDIR=Release
LINKOBJS=$(OBJS)
LINKLIBS=libJNIPPCore.lib

TARGETNAME=simpleDemo

"$(OUTDIR)":
        if not exist "$(OUTDIR)\$(NULL)" mkdir "$(OUTDIR)"

.cpp.obj:
```

```
        $(CPP) @<< $(CPPFLAGS) /Fo$(<:.cpp=.obj) $<
<<

!ENDIF

"$(OUTDIR)\$(TARGETNAME).$(EXT)" : "$(OUTDIR)" $(OBJS)
        $(LINK) @<< $(LINKFLAGS) /out:"$(OUTDIR)\$(TARGETNAME).$(EXT)" $(LINKOBJS) $(LINKLIBS)
<<

all : "$(OUTDIR)\$(TARGETNAME).$(EXT)"

clean::
        -@del $(LINKOBJS:/=\)
        -@rmdir /s /q $(OUTDIR)

rebuild : clean all
```

Notes:

1. The `TARGETTYPE` is determined by the associated `targetType` attribute of the input project. Because we specified an executable, the makefile is set up to build an executable image from the files in the project.
2. The `Main.cpp` file has been added to the list of sources to be compiled and linked into the target executable. The list of sources includes all of the input classes specified in the project and possibly more depending on the project settings (we will see this illustrated later).

Code Snip 3 - 6: The Generated NMakefile

```
INCLUDES= -I. -I$(JNIPP_HOME)/include -I$(JAVA_HOME)/include -I$(JAVA_HOME)/include/$(OS)
TARGETTYPE= consoleapp
TARGETNAME= simpleDemo

ifeq ($(TARGETTYPE), shlib)
DBGCPPFLAGS= -O0 -g -fPIC -c -Wall $(INCLUDES)
RELCPPFLAGS= -O0 -fPIC -c -Wall $(INCLUDES)
else
DBGCPPFLAGS= -O0 -g -c -Wall $(INCLUDES)
RELCPPFLAGS= -O0 -c -Wall $(INCLUDES)
endif

LIBPATH= -L$(JNIPP_HOME)/lib
DBGLIBS= $(LIBPATH) -lJNIPPCore_d
RELLIBS= $(LIBPATH) -lJNIPPCore

LINKFLAGS=
EXT=
ifeq ($(TARGETTYPE), shlib)
ifeq ($(OS), linux)
LINKFLAGS=-shared
EXT=.so
else
SHLIBCMD=-G
EXT=.so
endif
endif

%_d.o: %.cpp
                        g++ $(DBGCPPFLAGS) $< -o $@

%.o: %.cpp
                        g++ $(RELCPPFLAGS) $< -o $@

1:

SRCS=  .\demo\chapters\san\simple\SimpleDemoProxy.cpp
SRCS+= Main.cpp


DBGOBJS=$(patsubst %.cpp, %_d.o, $(SRCS))
RELOBJS=$(patsubst %.cpp, %.o, $(SRCS))

all:         Debug Release

Debug: dirs $(DBGOBJS)
                        g++ $(LINKFLAGS) -o Debug/$(TARGETNAME)_d$(EXT) $(DBGOBJS) $(DBGLIBS)

Release: dirs $(RELOBJS)
                        g++ $(LINKFLAGS) -o Release/$(TARGETNAME)$(EXT) $(RELOBJS) $(RELLIBS)

dirs: $(dummy)
                @mkdir -p Debug
```

```
                @mkdir -p Release

clean: $(dummy)
                        @rm -rf Debug
                        @rm -rf Release

rebuild: clean Debug Release
```

Notes:

1. The `Main.cpp` file has been added to the list of sources to be compiled and linked into the target executable. The list of sources includes all of the input classes specified in the project and possibly more depending on the project settings (we will see this illustrated later).

Code Snip 3 - 7: The Generated GNUMakefile

3.2.5 Putting the Generated Code to Use

Now that we have a C++ proxy class for the Java class that we wanted to expose on the C++ side of the JNI, we will put it to use with a simple `main( )` entry point that we can invoke from the command interpreter. The content of this file is shown below.

```
#include <iostream.h>
#include "net/sourceforge/jnipp/JVM.h"
#include "net/sourceforge/jnipp/JNIEnvHelper.h"
#include "demo/chapters/san/simple/SimpleDemoProxy.h"
#include "net/sourceforge/jnipp/BaseException.h"

using namespace net::sourceforge::jnipp;
using namespace demo::chapters::san::simple;

int main(int argc, char** argv)
{
        try
        {
1:
                /*
                 * Note that we are relying on the JVM class to resolve the location of the JVM
                 * through the JVM_HOME environment variable.  An exception will be thrown if it
                 * is not set.
                 */
                JVM::load();

2:
                JVM::load( jrePath );


3:
                SimpleDemoProxy sdp;

4:
                jclass cls = static_cast<jclass>( JNIEnvHelper::NewGlobalRef( JNIEn-
vHelper::FindClass( "java/lang/Integer" ) ) );
                jmethodID mid = JNIEnvHelper::GetMethodID( cls, "<init>", "(I)V" );
                jobject integerParam = JNIEnvHelper::NewGlobalRef( JNIEnvHelper::NewObject( cls, mid,
42 ) );

5:
                sdp.printValue( integerParam );
                JVM::unload();
```

```
        }
6:
        catch(BaseException& ex)
        {
                cerr << "caught exception: " << ex.getMessage().c_str() << endl;
        }
        return 0;
}
```

Notes:

1. The examples assume that the `JVM_HOME` environment variable is set according to the steps outlined in Chapter 2.
2. The `JVM` class is included in the core runtime library and is used to load and unload the virtual machine.
3. This constructor not only creates the C++ proxy instance, but also instantiates the associated Java class using its default constructor.
4. The parameter is manually created by locating the class and method, then using them to create an instance of the `java.lang.Integer` class with the constructor that takes a single integer parameter. We will soon see the use of a generated proxy class to simplify the parameter passing.j
5. And here is the call to our single method, `printValue()`. Once finished, we unload the JVM using the `JVM` helper class.
6. The `BaseException` class can be utilized to catch any exceptions generated by the core runtime library. It is not used, however, to catch exceptions thrown by the JVM.

Code Snip 3 - 8: Exercising the C++ Proxy Class

Note the comments regarding the invocation of the JVM using the `JVM::load()` method call. This method has two forms, one that accepts a string parameter specifying the JVM shared library to be loaded, and the one shown in the examples that accepts no arguments. This form of the method expects the `JVM_HOME` environment variable to be set such that the JVM shared library (and its dependencies) can be located and loaded. If the variable is not set, then an exception will be thrown. See Chapter 2 for details regarding your platform.

3.2.6 Building the Project

With the help of the generated makefiles, the module is fairly simple to build. The makefiles rely on the `JAVA_HOME` and `JNIPP_HOME` environment variables to resolve includes and libraries. The generated makefiles also assume that they are being invoked from the same directory in which they reside. The following is a sample run on a Win32 platform.

```
C:\development\jnipp\demo\chapters\san\simple>nmake /f simple.mak CFG="Debug" all

Microsoft (R) Program Maintenance Utility   Version 6.00.8168.0
Copyright (C) Microsoft Corp 1988-1998. All rights reserved.

        if not exist "Debug\" mkdir "Debug"
        cl.exe @C:\DOCUME~1\ptrewhel\LOCALS~1\Temp\nma01032. /nologo /GX /W3 /FD /D "WIN32" /D
"_WINDOWS" /D "_MBCS" /I .\ /I "c:\java\jdk1.3/include" /I "c:\java\jdk1.3/include/win32" /I
"c:\development\jnipp\dist/include" /c /MDd /Zi /Od /D "_DEBUG"
/Fo.\demo\chapters\san\simple\SimpleDemoProxy_d.obj .\demo\chapters\san\simple\SimpleDemoProxy.cpp
SimpleDemoProxy.cpp
        cl.exe @C:\DOCUME~1\ptrewhel\LOCALS~1\Temp\nmb01032. /nologo /GX /W3 /FD /D "WIN32" /D
"_WINDOWS" /D "_MBCS" /I .\ /I "c:\java\jdk1.3/include" /I "c:\java\jdk1.3/include/win32" /I
"c:\development\jnipp\dist/include" /c /MDd /Zi /Od /D "_DEBUG" /FoMain_d.obj Main.cpp
Main.cpp
        link.exe @C:\DOCUME~1\ptrewhel\LOCALS~1\Temp\nmc01032. /nologo /machine:IX86
/libpath:"c:\development\jnipp\dist/lib" /subsystem:console /incremental:no /debug
/out:"Debug\simpleDemo_d."exe"" .\demo\chapters\san\simple\SimpleDemoProxy_d.obj Main_d.obj
libJNIPPCore_d.lib

C:\development\jnipp\demo\chapters\san\simple>
```

Output Sample 3 - 3: Building the Project

The process is similar for other platforms. If you get an unsatisfied symbol error for `main()`, then you probably forgot to append the `Main.cpp` file to the list of input sources in the generated makefile. See the earlier section that covered the generated files.

3.2.7 Running the Executable

Before we can run the executable module, we must ensure that we can resolve the `libJNIPPCore` shared library and that the target Java class is in the `CLASSPATH`. In a Win32 environment, the shared library must either be in the `PATH` or the working directory, while the various UNIX flavors typically consult the `LD_LIBRARY_PATH` and the working directory. See Chapter 2 for further information on setting environment variables for your platform. Below is a sample run in a Win32 environment.

```
C:\development\jnipp\demo\chapters\san>set PATH=%PATH%;%JNIPP_HOME%\lib

C:\development\jnipp\demo\chapters\san>set CLASSPATH=%CLASSPATH%;%JNIPP_HOME%\build\classes

C:\development\jnipp\demo\chapters\san>simple\Debug\simpleDemo_d
value is 42

C:\development\jnipp\demo\chapters\san>
```

Output Sample 3 - 4: Running the Example

3.2.8 A Peek Under the Hood

So how does it all work? The generated C++ proxy class creates an instance of its Java peer on the other side of the JNI during a constructor call. Because we did not provide a constructor for our simple Java class, it was provided a default by the Java compiler. For each of the constructors of the input Java class, there is a corresponding constructor for the generated C++ proxy class. The C++ proxy class makes a `NewObject()` call in its constructors and passes in the appropriate parameters (in this case none). It then saves the `jobject` return value for use in subsequent method calls.

All of the rest is pretty straightforward, as illustrated in the following sequence diagram. Each of the methods of the input Java class results in a corresponding method in the generated C++ proxy class. When one of the generated proxy methods is called, the `methodID` is retrieved and utilized in the subsequent call to the corresponding method of the Java class instance.

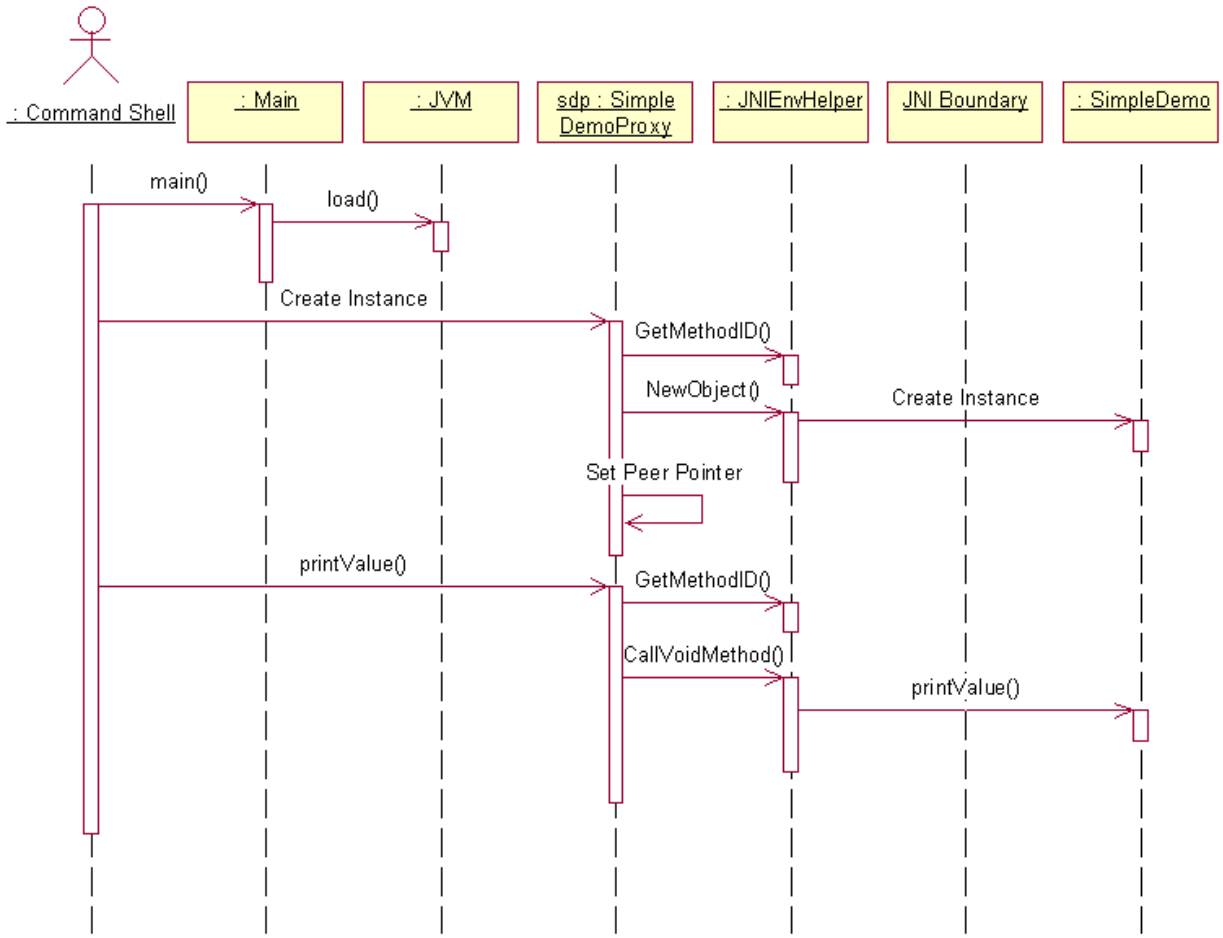

Figure 3 - 3: Sequence of Events for the Simple Demo

3.3 Accessing and Manipulating Fields

The JNI++ Proxy Generator can be used to generate field accessor and mutator methods for the target Java classes, providing the ability to directly modify the attributes of the target Java class. This section will illustrate this capability with an appropriate example.

3.3.1 The Input Java Class

For this example we will define a simple class that contains a single `String` field and a constructor that accepts a `String` argument to initialize the field. The source is shown below.

```
package demo.chapters.san.fields;

public class FieldsDemo
{
        protected String stringField;

        public FieldsDemo(String value)
```

```
        {
                stringField = value;
        }
}
```

Code Snip 3 - 9: The Input Java Class

3.3.2 The Project File

Setting the `attributeGetters` attribute to "true" will result in field accessor methods being generated in the C++ proxy class for each of the fields of the input Java class. Similarly, setting the `attributeSetters` attribute to "true" will result in field mutator methods. Here we set them both.

```
<?xml version="1.0"?>

<project name="fields" targetType="exe" targetName="fieldsDemo">

1:      <proxygen useRichTypes="false" attributeGetters="true" attributeSetters="true">

                <input-classes>
                        <input-class name="demo.chapters.san.fields.FieldsDemo"/>
                </input-classes>
        </proxygen>
        <nmakefile name="fields.mak"/>
        <gnumakefile name="Makefile"/>
</project>
```

Notes:

1. The `proxygen` element specifies that attribute accessor and mutator methods are to be generated for the input Java classes.

Code Snip 3 - 10: The Project File

3.3.3 Invoking the Code Generator

Beginning with this example, we will be using the Ant build tool to compile the input Java class, invoke the code generator, build the project and finally run it. The build file for this project is `fields.xml`, located in the `JNIPP_HOME/demo/chapters/san` directory. Launch Ant from this directory, pass in the name of the build file and specify to build target "generate", as shown below.

```
C:\development\jnipp\demo\chapters\san>java org.apache.tools.ant.Main -buildfile fields.xml generate
Buildfile: fields.xml

prepare:

compile:
    [javac] Compiling 1 source file to C:\development\jnipp\build\classes

generate:
     [java] generating C++ Proxy Class for demo.chapters.san.fields.FieldsDemo ...
     [java] generating NMakefile for project fields ...
     [java] generating GNU Makefile for project fields ...
     [java]
     [java] code generation complete.

BUILD SUCCESSFUL

Total time: 3 seconds

C:\development\jnipp\demo\chapters\san>
```

Output Sample 3 - 5: Invoking the Code Generator with Ant

3.3.4 A Quick Look at the C++ Proxy Class

As expected, the code generator has created a single C++ proxy class with a total of three source files. In this section we will highlight the relevant portions of the generated C++ proxy class.

```
#ifndef __demo_chapters_san_fields_FieldsDemoProxy_H
#define __demo_chapters_san_fields_FieldsDemoProxy_H


#include <jni.h>
#include <string>


namespace demo
{
        namespace chapters
        {
                namespace san
                {
                        namespace fields
                        {
                                class FieldsDemoProxy
                                {
                                private:
                                        static std::string className;
                                        static jclass objectClass;
                                        jobject peerObject;

                                protected:
                                        FieldsDemoProxy(void* unused);
                                        virtual jobject _getPeerObject() const;

                                public:
                                        static jclass _getObjectClass();
                                        static inline std::string _getClassName()
                                        {
                                                return className;
                                        }

                                        jclass getObjectClass();
                                        operator jobject();
                                        // constructors
                                        FieldsDemoProxy(jobject obj);
1:
                                        FieldsDemoProxy(jstring p0);

                                        virtual ~FieldsDemoProxy();
                                        FieldsDemoProxy& operator=(const FieldsDemoProxy& rhs);
2:
                                        // attribute getters
                                        jstring getStringField() const;
                                        // attribute setters
                                        void setStringField(jstring stringField);


                                        // methods

                                };
                        }
                }
        }
}

#endif
```

Notes:

1. We provided a single constructor in the input Java class that accepts a `String` argument, and here is its C++ counterpart.
2. Here are the generated field accessor and mutator methods for the single field in the input Java class.

Code Snip 3 - 11: The C++ Proxy Header File

```
#include "net/sourceforge/jnipp/JNIEnvHelper.h"
#include "FieldsDemoProxy.h"

using namespace net::sourceforge::jnipp;
using namespace demo::chapters::san::fields;


std::string FieldsDemoProxy::className = "demo/chapters/san/fields/FieldsDemo";
jclass FieldsDemoProxy::objectClass = NULL;

jclass FieldsDemoProxy::_getObjectClass()
{
        if ( objectClass == NULL )
                objectClass = static_cast&lt;jclass&gt;( JNIEnvHelper::NewGlobalRef( JNIEn-
vHelper::FindClass( className.c_str() ) ) );

        return objectClass;
}

FieldsDemoProxy::FieldsDemoProxy(void* unused)
{
}

jobject FieldsDemoProxy::_getPeerObject() const
{
        return peerObject;
}

jclass FieldsDemoProxy::getObjectClass()
{
        return _getObjectClass();
}

FieldsDemoProxy::operator jobject()
{
        return _getPeerObject();
}

// constructors
FieldsDemoProxy::FieldsDemoProxy(jobject obj)
{
        peerObject = JNIEnvHelper::NewGlobalRef( obj );
}

FieldsDemoProxy::FieldsDemoProxy(jstring p0)
{
        jmethodID mid = JNIEnvHelper::GetMethodID( _getObjectClass(), "&lt;init&gt;",
 "(Ljava/lang/String;)V" );
        peerObject = JNIEnvHelper::NewGlobalRef( JNIEnvHelper::NewObject( _getObjectClass(), mid, p0
) );
}

FieldsDemoProxy::~FieldsDemoProxy()
{
        JNIEnvHelper::DeleteGlobalRef( peerObject );
}

FieldsDemoProxy& FieldsDemoProxy::operator=(const FieldsDemoProxy& rhs)
{
        JNIEnvHelper::DeleteGlobalRef( peerObject );
        peerObject = JNIEnvHelper::NewGlobalRef( rhs.peerObject );
        return *this;
}

// attribute getters

1:

jstring FieldsDemoProxy::getStringField() const
{
        jfieldID fid = JNIEnvHelper::GetFieldID( _getObjectClass(), "stringField",
 "Ljava/lang/String;" );
        return reinterpret_cast&lt; jstring &gt;( JNIEnvHelper::GetObjectField( _getPeerObject(), fid
) );
}
```

```
   // attribute setters

2:

   void FieldsDemoProxy::setStringField(jstring stringField)
   {
           jfieldID fid = JNIEnvHelper::GetFieldID( _getObjectClass(), "stringField",
    "Ljava/lang/String;" );
           JNIEnvHelper::SetObjectField( _getPeerObject(), fid, stringField );
   }


   // methods
```

Notes:

1. The implemenation of generated attribute accessor methods are all similar. They differ only in the name and type of the field retrieved in the `GetFieldID()` call and the call to retrieve the value, which is dependent on its type.
2. Similarly, the generated attribute mutator methods differ only in the field id retrieved and the subsequent call to set the value, which is also dependent on its type.

Code Snip 3 - 12: The C++ Proxy Implementation File

3.3.5 Using the Generated Proxy Class

Again, we write a simple `main()` method to exercise the generated proxy class. This file is similar to that presented in the previous section. It loads the JVM in a similar fashion, creates a local proxy variable and then exercises the attribute accessors and mutators.

```
   #include <iostream.h>
   #include "net/sourceforge/jnipp/JVM.h"
   #include "net/sourceforge/jnipp/JNIEnvHelper.h"

1:

   #include "net/sourceforge/jnipp/JStringHelper.h"

   #include "demo/chapters/san/fields/FieldsDemoProxy.h"
   #include "net/sourceforge/jnipp/BaseException.h"

   using namespace net::sourceforge::jnipp;
   using namespace demo::chapters::san::fields;

   int main(int argc, char** argv)
   {
           try
           {
                   /*
                    * Note that we are relying on the JVM class to resolve the location of the JVM
                    * through the JVM_HOME environment variable.  An exception will be thrown if it
                    * is not set.
                    */
                   JVM::load();

2:

                   FieldsDemoProxy fdp( static_cast<jstring>( JStringHelper( "Initial Value" ) ) );


3:

                   cout << "Before modifying, the value is \"" << JStringHelper( fdp.getStringField() )
   << "\"" << endl;
                   fdp.setStringField( JStringHelper( "New Value" ) );
                   cout << "After modifying, the value is \"" << JStringHelper( fdp.getStringField() )
   << "\"" << endl;

                   JVM::unload();
           }
           catch(BaseException& ex)
```

```
        {
                cerr << "caught exception: " << ex.getMessage().c_str() << endl;
        }

        return 0;
}
```

Notes:

1. Here we include the header for the `JStringHelper` class, one of many helper classes at our disposal.
2. Here is the `JStringHelper` class in action. We create an instance of this helper class to pass a `String` to the target Java class constructor. JNI++ provides several helper classes to ease the burden of dealing with the native JNI data types. These are covered in a later chapter.
3. Here the value is simply printed out using the generated attribute accessor method, then modified with the generated attribute mutator method, then printed out again using the attribute accessor. Note the use of the `JStringHelper` class throughout.

Code Snip 3 - 13: The Driver Code

3.3.6 Building and Running

Once again, we invoke the Ant build tool, but this time specify the "native" target. This will build the Debug and Release configurations of the project. Don't forget to add the `Main.cpp` file to the list of sources (this is, of course, already done if you are using the files supplied in the distribution).

```
C:\development\jnipp\demo\chapters\san>java org.apache.tools.ant.Main -buildfile fields.xml native
Buildfile: fields.xml

prepare:

compile:
    [javac] Compiling 1 source file to C:\development\jnipp\build\classes

generate:
     [java] generating C++ Proxy Class for demo.chapters.san.fields.FieldsDemo ...
     [java] generating NMakefile for project fields ...
     [java] generating GNU Makefile for project fields ...
     [java]
     [java] code generation complete.

native:
     [exec]     if not exist "Debug\" mkdir "Debug"
     [exec]
     [exec] Microsoft (R) Program Maintenance Utility   Version 6.00.8168.0
     [exec] Copyright (C) Microsoft Corp 1988-1998. All rights reserved.
     [exec]
     [exec]     cl.exe @C:\DOCUME~1\ptrewhel\LOCALS~1\Temp\nma00824. /nologo /GX /W3 /FD /D "WIN32" /D
"_WINDOWS" /D "_MBCS" /I .\ /I "c:\java\jdk1.3/include" /I "c:\java\jdk1.3/include/win32" /I
"c:\development\jnipp\dist/include" /c /MDd /Zi /Od /D "_DEBUG"
/Fo.\demo\chapters\san\fields\FieldsDemoProxy_d.obj .\demo\chapters\san\fields\FieldsDemoProxy.cpp
     [exec] FieldsDemoProxy.cpp
     [exec]     cl.exe @C:\DOCUME~1\ptrewhel\LOCALS~1\Temp\nmb00824. /nologo /GX /W3 /FD /D "WIN32" /D
"_WINDOWS" /D "_MBCS" /I .\ /I "c:\java\jdk1.3/include" /I "c:\java\jdk1.3/include/win32" /I
"c:\development\jnipp\dist/include" /c /MDd /Zi /Od /D "_DEBUG" /FoMain_d.obj Main.cpp
     [exec] Main.cpp
     [exec]     link.exe @C:\DOCUME~1\ptrewhel\LOCALS~1\Temp\nmc00824. /nologo /machine:IX86
/libpath:"c:\development\jnipp\dist/lib" /subsystem:console /incremental:no /debug
/out:"Debug\fieldsDemo_d."exe"" .\demo\chapters\san\fields\FieldsDemoProxy_d.obj Main_d.obj
libJNIPPCore_d.lib
     [exec]     if not exist "Release\" mkdir "Release"
     [exec]
     [exec] Microsoft (R) Program Maintenance Utility   Version 6.00.8168.0
     [exec] Copyright (C) Microsoft Corp 1988-1998. All rights reserved.
     [exec]
     [exec]     cl.exe @C:\DOCUME~1\ptrewhel\LOCALS~1\Temp\nma00308. /nologo /GX /W3 /FD /D "WIN32" /D
"_WINDOWS" /D "_MBCS" /I .\ /I "c:\java\jdk1.3/include" /I "c:\java\jdk1.3/include/win32" /I
"c:\development\jnipp\dist/include" /c /MD /O2 /D "NDEBUG"
/Fo.\demo\chapters\san\fields\FieldsDemoProxy.obj .\demo\chapters\san\fields\FieldsDemoProxy.cpp
     [exec] FieldsDemoProxy.cpp
     [exec]     cl.exe @C:\DOCUME~1\ptrewhel\LOCALS~1\Temp\nmb00308. /nologo /GX /W3 /FD /D "WIN32" /D
"_WINDOWS" /D "_MBCS" /I .\ /I "c:\java\jdk1.3/include" /I "c:\java\jdk1.3/include/win32" /I
"c:\development\jnipp\dist/include" /c /MD /O2 /D "NDEBUG" /FoMain.obj Main.cpp
     [exec] Main.cpp
     [exec]     link.exe @C:\DOCUME~1\ptrewhel\LOCALS~1\Temp\nmc00308. /nologo /machine:IX86
```

```
/libpath:"c:\development\jnipp\dist/lib" /subsystem:console /incremental:no
/out:"Release\fieldsDemo."exe"" .\demo\chapters\san\fields\FieldsDemoProxy.obj Main.obj libJNIPPCore.lib
     [copy] Copying 2 files to C:\development\jnipp\build\demo\chapters\san\fields

BUILD SUCCESSFUL

Total time: 6 seconds

C:\development\jnipp\demo\chapters\san>
```

Output Sample 3 - 6: Using Ant to Build the Executable

Running the newly-built executable delivers the expected results as shown below.

```
C:\development\jnipp\demo\chapters\san>java org.apache.tools.ant.Main -buildfile fields.xml run
Buildfile: fields.xml

run:
     [exec] Before modifying, the value is "Initial Value"
     [exec] After modifying, the value is "New Value"

BUILD SUCCESSFUL

Total time: 1 second

C:\development\jnipp\demo\chapters\san>
```

Output Sample 3 - 7: The Final Result

3.4 Method Name Disambiguation

When using native JNI data types as parameter values, problems can occur when an input class has two methods or constructors that have the same name but differ in the parameter types. Although the parameter types differ in Java, their native JNI data type representation may be the same, resulting in two or more generated methods that have the same name and identical signatures.

The JNI++ code generators attempt to resolve this problem by applying logic to disambiguate the method names. The algorithm is simple -- if two methods are being generated that have the same name and identical JNI signatures, then the second method is disambiguated by appending an incremental integer value. Note that this only occurs when the project file specifies the use of only native JNI data types. As we will later see, the problem is far more elegantly solved when using the "rich types" feature of the JNI++ code generators.

3.4.1 The Input Java Class

This class has been purposefully composed to have two methods with the same name but different Java parameter types. As we will soon see upon code generation, this would cause a problem without method disambiguation.

```
package demo.chapters.san.disambiguate;

public class DisambiguateDemo
{
        public void printValue(Integer value)
        {
```

```
                System.out.println( "value is " + value );
        }

        public void printValue(Double value)
        {
                System.out.println( "value is " + value );
        }
}
```

Code Snip 3 - 14: The Input Java Class

3.4.2 The Project File

The method name disambiguation mechanism is employed automatically to resolve method name conflicts unless the "rich type generation" option is set (which we will see shortly). The project file below includes no new options to turn on the disambiguation feature.

```
<?xml version="1.0"?>

<project name="disambiguate" targetType="exe" targetName="disambiguateDemo">
        <proxygen useRichTypes="false"
                <input-classes>
                        <input-class name="demo.chapters.san.disambiguate.DisambiguateDemo"/>
1:
                        <input-class name="java.lang.Integer"/>
                        <input-class name="java.lang.Double"/>

                </input-classes>
        </proxygen>
        <nmakefile name="disambiguate.mak"/>
        <gnumakefile name="Makefile"/>
</project>
```

Notes:

1. For convenience, we will also generate C++ proxy classes for the `Integer` and `Double` parameter types.

Code Snip 3 - 15: The Project File

3.4.3 Invoking the Code Generator

Using the Ant build tool, the proxy classes are generated as in the previous example by passing in the `disambiguate.xml` file and specifying the "generate" target. C++ proxy classes are generated for each of the three input Java class files specified in the project. The output for this and the following examples will be omitted unless there are interesting details to be illustrated.

3.4.4 The Disambiguated Methods

This section will highlight the code relevant to the disambiguated methods. Note that although proxy classes were also generated for the `Integer` and `Double` Java classes, they are not relevant to this discussion and will therefore not be reviewed.

```
#ifndef __demo_chapters_san_disambiguate_DisambiguateDemoProxy_H
#define __demo_chapters_san_disambiguate_DisambiguateDemoProxy_H
```

```
#include <jni.h>
#include <string>


namespace demo
{
        namespace chapters
        {
                namespace san
                {
                        namespace disambiguate
                        {
                                class DisambiguateDemoProxy
                                {
                                private:
                                        static std::string className;
                                        static jclass objectClass;
                                        jobject peerObject;

                                protected:
                                        DisambiguateDemoProxy(void* unused);
                                        virtual jobject _getPeerObject() const;

                                public:
                                        static jclass _getObjectClass();
                                        static inline std::string _getClassName()
                                        {
                                                return className;
                                        }

                                        jclass getObjectClass();
                                        operator jobject();
                                        // constructors
                                        DisambiguateDemoProxy(jobject obj);
                                        DisambiguateDemoProxy();

                                        virtual ~DisambiguateDemoProxy();
                                        DisambiguateDemoProxy& operator=(const DisambiguateDemoProxy&
rhs);

                                        // methods
                                        /*
                                         * void printValue(Double);
                                         */
                                        void printValue(jobject p0);

1:                                      /*
                                         * void printValue(Integer);
                                         */

                                        void printValue1(jobject p0);


                                };
                        }
                }
        }
}

#endif
```

Notes:

1. Here is the the method that was chosen for disambiguation. Note the integer value appended to the end of the method name. Without this, it would be identical to the previous method.

Code Snip 3 - 16: The Generated Proxy Header

```
#include "net/sourceforge/jnipp/JNIEnvHelper.h"
#include "DisambiguateDemoProxy.h"

using namespace net::sourceforge::jnipp;
using namespace demo::chapters::san::disambiguate;


std::string DisambiguateDemoProxy::className = "demo/chapters/san/disambiguate/DisambiguateDemo";
jclass DisambiguateDemoProxy::objectClass = NULL;
```

```
jclass DisambiguateDemoProxy::_getObjectClass()
{
        if ( objectClass == NULL )
                objectClass = static_cast<jclass>( JNIEnvHelper::NewGlobalRef( JNIEn-
vHelper::FindClass( className.c_str() ) ) );

        return objectClass;
}

DisambiguateDemoProxy::DisambiguateDemoProxy(void* unused)
{
}

jobject DisambiguateDemoProxy::_getPeerObject() const
{
        return peerObject;
}

jclass DisambiguateDemoProxy::getObjectClass()
{
        return _getObjectClass();
}

DisambiguateDemoProxy::operator jobject()
{
        return _getPeerObject();
}

// constructors
DisambiguateDemoProxy::DisambiguateDemoProxy(jobject obj)
{
        peerObject = JNIEnvHelper::NewGlobalRef( obj );
}

DisambiguateDemoProxy::DisambiguateDemoProxy()
{
        jmethodID mid = JNIEnvHelper::GetMethodID( _getObjectClass(), "<init>", "()V" );
        peerObject = JNIEnvHelper::NewGlobalRef( JNIEnvHelper::NewObject( _getObjectClass(), mid ) );
}

DisambiguateDemoProxy::~DisambiguateDemoProxy()
{
        JNIEnvHelper::DeleteGlobalRef( peerObject );
}

DisambiguateDemoProxy& DisambiguateDemoProxy::operator=(const DisambiguateDemoProxy& rhs)
{
        JNIEnvHelper::DeleteGlobalRef( peerObject );
        peerObject = JNIEnvHelper::NewGlobalRef( rhs.peerObject );
        return *this;
}

// methods
void DisambiguateDemoProxy::printValue(jobject p0)
{
        static jmethodID mid = NULL;
        if ( mid == NULL )
                mid = JNIEnvHelper::GetMethodID( _getObjectClass(), "printValue",
 "(Ljava/lang/Double;)V" );
        JNIEnvHelper::CallVoidMethod( _getPeerObject(), mid, p0 );
}
```

1:

```
void DisambiguateDemoProxy::printValue1(jobject p0)
{
        static jmethodID mid = NULL;
        if ( mid == NULL )
                mid = JNIEnvHelper::GetMethodID( _getObjectClass(), "printValue",
 "(Ljava/lang/Integer;)V" );
        JNIEnvHelper::CallVoidMethod( _getPeerObject(), mid, p0 );
}
```

Notes:

1. Note that the implementation of the disambiguated method looks up a different method id. Other than this difference (and the disambiguated name, of course), the methods are identical.

Code Snip 3 - 17: The Generated Proxy Implementation

3.4.5 Using the Generated Proxy Classes

Our `main()` method below simply makes calls to each of the disambiguated methods, passing in a parameter appropriate for the method.

```
   #include <iostream.h>
   #include "net/sourceforge/jnipp/JVM.h"
   #include "net/sourceforge/jnipp/JNIEnvHelper.h"
   #include "demo/chapters/san/disambiguate/DisambiguateDemoProxy.h"

1:

   #include "java/lang/IntegerProxy.h"
   #include "java/lang/DoubleProxy.h"

   #include "net/sourceforge/jnipp/BaseException.h"

   using namespace net::sourceforge::jnipp;
   using namespace demo::chapters::san::disambiguate;
   using namespace java::lang;

   int main(int argc, char** argv)
   {
           try
           {
                   /*
                    * Note that we are relying on the JVM class to resolve the location of the JVM
                    * through the JVM_HOME environment variable.  An exception will be thrown if it
                    * is not set.
                    */
                   JVM::load();

                   DisambiguateDemoProxy ddp;

2:

                   ddp.printValue( DoubleProxy( 123.456 ) );
                   ddp.printValue1( IntegerProxy( 123 ) );


                   JVM::unload();
           }
           catch(BaseException& ex)
           {
                   cerr << "caught exception: " << ex.getMessage().c_str() << endl;
           }

           return 0;
   }
```

Notes:

1. Here we include the headers for the generated parameter type proxy classes.
2. Here are the calls to the generated methods. Note the use of the generated `DoubleProxy` and `IntegerProxy` classes. Also note that, although the methods have been disambiguated, it is still unclear without viewing the generated source which method accepts what parameters.

Code Snip 3 - 18: The Driver Code

3.4.6 Building and Running

Using the same buildfile and specifying the "native" target, the debug and release modules are built. From this example forward, the output of the build process will be omitted to save space.

With the newly-built executable, the test run can be invoked by specifying the "run" target using the supplied Ant build file. The output, shown below, shows no surprises.

```
C:\development\jnipp\demo\chapters\san>java org.apache.tools.ant.Main -buildfile disambiguate.xml run
Buildfile: disambiguate.xml

run:
     [exec] value is 123.456
     [exec] value is 123

BUILD SUCCESSFUL

Total time: 1 second

C:\development\jnipp\demo\chapters\san>
```

Output Sample 3 - 8: Running the Example

Note that although method name disambiguation will prevent method name clashes in the generated C++ proxy code, there is currently no mechanism, other than the use of "rich types", for disambiguating constructors. The constructors obviously cannot be named differently and still behave as constructors. If the example had shown the two input methods as constructors, the code would not compile. The only option at that point is to use the "rich types" code generation feature. This will be covered in the last section of the chapter.

Another disadvantage with this approach is that it is not obvious by examining the method signatures which one takes a `java.lang.Integer` and which takes a `java.lang.Double` argument. The only way to really find out is to either look at the generated comment above the methods in the header file or examine the call to get the field id in the implementation. This problem also goes away when using "rich types".

3.5 Generating an Inheritance Tree

The JNI++ Proxy Generator is capable of generating C++ proxy classes that mimic the inheritance hierarchy of the associated Java classes. With this option selected, the code generator will generate C++ proxy classes not only for the input classes, but also for any superclasses all the way up the tree. If the input Java class is an interface, then proxy classes are generated for the "implements" hierarchy all the way up the tree. In addition to generating the proxy classes, the class definitions are modified to reflect the inheritance tree.

3.5.1 The Input Java Class

This example defines two Java classes. The `InheritanceDemoBase` class serves as the superclass for the `InheritanceDemo` class. Both classes define a `printMessage()` method which simply prints a message to the console for illustration. The `InheritanceDemoBase` class also defines a `printBaseMessage()` method. The classes are shown in their entirety below.

```
package demo.chapters.san.inheritance;

public class InheritanceDemoBase
{
        public void printBaseMessage()
        {
                System.out.println( "Hello from InheritanceDemoBase" );
        }

        public void printMessage()
        {
                printBaseMessage();
        }
}
```

Code Snip 3 - 19: The Base Class

```
package demo.chapters.san.inheritance;

public class InheritanceDemo
        extends InheritanceDemoBase
{
        public void printMessage()
        {
                System.out.println( "Hello from InheritanceDemo" );
        }
}
```

Code Snip 3 - 20: The Derived Class

3.5.2 The Project File

```
<?xml version="1.0"?>

<project name="inheritance" targetType="exe" targetName="inheritanceDemo">

1:      <proxygen useRichTypes="false" useInheritance="true">

                <input-classes>
                        <input-class name="demo.chapters.san.inheritance.InheritanceDemo"/>
                </input-classes>
        </proxygen>
        <nmakefile name="inheritance.mak"/>
        <gnumakefile name="Makefile"/>
</project>
```

Notes:

1. Setting the `useInheritance` flag to "true" will trigger code with inheritance constructs to be generated.

Code Snip 3 - 21: The Project File

3.5.3 Generating the C++ Proxy Classes

Using the Ant build file, the code is generated in the usual manner by specifying the "generate" target. Note that two proxy classes are generated that were not specified in the project file. These proxy classes are generated because their Java counterparts are in the inheritance hierarchy of the input Java class. The proxy classes must be generated for these classes because the generated proxy for the input Java class will derive from them. Note that a proxy class is *always* generated for `java.lang.Object`, because this is ultimately the root of any inheritance hierarchy.

3.5.4 A Quick Look at the Generated Code

The JNI++ Proxy Generator has generated proxy classes for not only the input `InheritanceDemo` class, but also for the `InheritanceDemoBase` superclass and its superclass, `java.lang.Object`. The following diagram illustrates the inheritance hierarchy of the Java classes compared with the generated proxy classes.

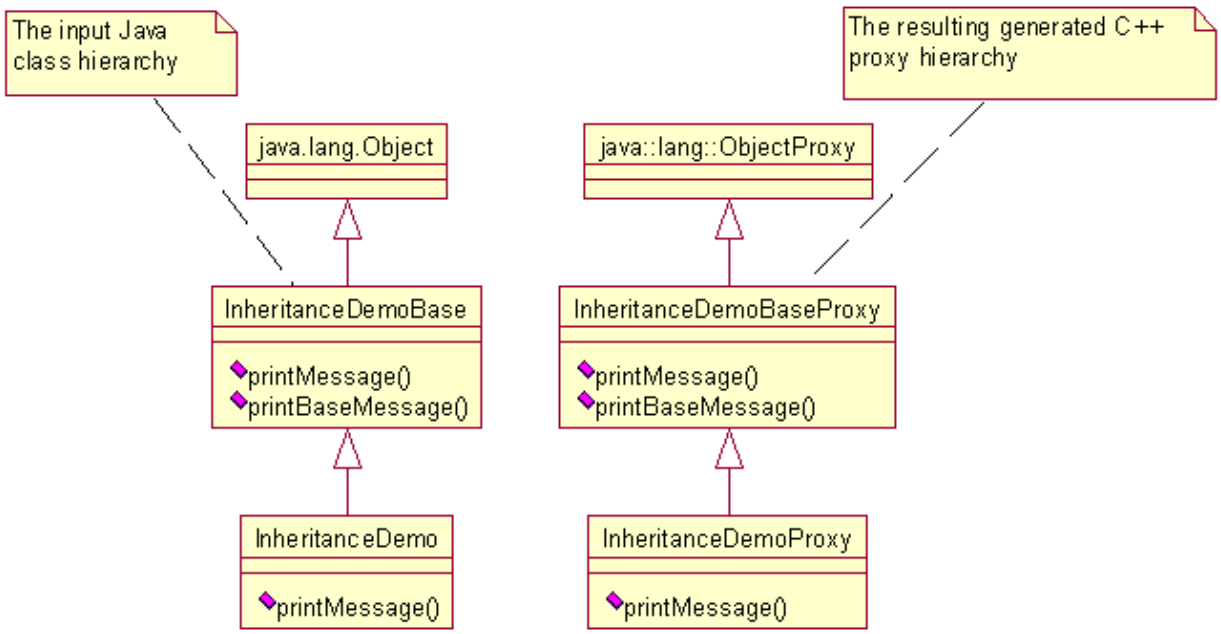

Figure 3 - 4: Parallel Worlds

3.5.4.1 The Generated Base Proxy Header

```
#ifndef __demo_chapters_san_inheritance_InheritanceDemoBaseProxy_H
#define __demo_chapters_san_inheritance_InheritanceDemoBaseProxy_H


#include <jni.h>
#include <string>

#include "java\lang\ObjectProxy.h"

namespace demo
{
        namespace chapters
        {
                namespace san
                {
                        namespace inheritance
                        {
1:
                                class InheritanceDemoBaseProxy : public java::lang::ObjectProxy

                                {
                                private:
                                        static std::string className;
                                        static jclass objectClass;
                                        jobject peerObject;

                                protected:
2:
                                        InheritanceDemoBaseProxy(void* unused);

                                        virtual jobject _getPeerObject() const;

                                public:
                                        static jclass _getObjectClass();
                                        static inline std::string _getClassName()
                                        {
                                                return className;
```

```
                                        }

                                        jclass getObjectClass();
                                        operator jobject();
                                        // constructors
                                        InheritanceDemoBaseProxy(jobject obj);
                                        InheritanceDemoBaseProxy();

                                        virtual ~InheritanceDemoBaseProxy();
                                        InheritanceDemoBaseProxy& operator=(const InheritanceDe-
moBaseProxy& rhs);

                                        // methods
                                        /*
                                         * void printBaseMessage();
                                         */
3:
                                        void printBaseMessage();

                                        /*
                                         * void printMessage();
                                         */
                                        void printMessage();

                                };
                        }
                }
        }
}

#endif
```

Notes:

1. As promised, the generated proxy class for `java.lang.Object` sits at the root of the tree.
2. This silly-looking constructor introduced in the first example will finally be used. The purpose of this constructor is to provide derived classes a mechanism for initializing the base class.
3. The `printBaseMessage()` method is defined only in the base class.

Code Snip 3 - 22: The Base Proxy Header

3.5.4.2 The Generated Base Proxy Implementation

```
#include "net/sourceforge/jnipp/JNIEnvHelper.h"
#include "InheritanceDemoBaseProxy.h"

using namespace net::sourceforge::jnipp;
using namespace demo::chapters::san::inheritance;


std::string InheritanceDemoBaseProxy::className =
 "demo/chapters/san/inheritance/InheritanceDemoBase";
jclass InheritanceDemoBaseProxy::objectClass = NULL;

jclass InheritanceDemoBaseProxy::_getObjectClass()
{
        if ( objectClass == NULL )
                objectClass = static_cast<jclass>( JNIEnvHelper::NewGlobalRef( JNIEn-
vHelper::FindClass( className.c_str() ) ) );

        return objectClass;
}

1:
InheritanceDemoBaseProxy::InheritanceDemoBaseProxy(void* unused)
        : java::lang::ObjectProxy( unused )


{
}
```

```
jobject InheritanceDemoBaseProxy::_getPeerObject() const
{
        return peerObject;
}

jclass InheritanceDemoBaseProxy::getObjectClass()
{
        return _getObjectClass();
}

InheritanceDemoBaseProxy::operator jobject()
{
        return _getPeerObject();
}

// constructors
InheritanceDemoBaseProxy::InheritanceDemoBaseProxy(jobject obj)
        : java::lang::ObjectProxy( reinterpret_cast<void*>(NULL) )

{
        peerObject = JNIEnvHelper::NewGlobalRef( obj );
}

InheritanceDemoBaseProxy::InheritanceDemoBaseProxy()
        : java::lang::ObjectProxy( reinterpret_cast<void*>(NULL) )

{
        jmethodID mid = JNIEnvHelper::GetMethodID( _getObjectClass(), "<init>", "()V" );
        peerObject = JNIEnvHelper::NewGlobalRef( JNIEnvHelper::NewObject( _getObjectClass(), mid ) );
}

InheritanceDemoBaseProxy::~InheritanceDemoBaseProxy()
{
        JNIEnvHelper::DeleteGlobalRef( peerObject );
}

InheritanceDemoBaseProxy& InheritanceDemoBaseProxy::operator=(const InheritanceDemoBaseProxy& rhs)
{
        JNIEnvHelper::DeleteGlobalRef( peerObject );
        peerObject = JNIEnvHelper::NewGlobalRef( rhs.peerObject );
        return *this;
}

// methods
void InheritanceDemoBaseProxy::printBaseMessage()
{
        static jmethodID mid = NULL;
        if ( mid == NULL )
                mid = JNIEnvHelper::GetMethodID( _getObjectClass(), "printBaseMessage", "()V" );
        JNIEnvHelper::CallVoidMethod( _getPeerObject(), mid );
}

void InheritanceDemoBaseProxy::printMessage()
{
        static jmethodID mid = NULL;
        if ( mid == NULL )
                mid = JNIEnvHelper::GetMethodID( _getObjectClass(), "printMessage", "()V" );
        JNIEnvHelper::CallVoidMethod( _getPeerObject(), mid );
}
```

Notes:

1. As mentioned in the annotations of the header file, the `void*` constructor is called for any superclasses for initialization. Case in point.

Code Snip 3 - 23: The Base Proxy Implementation

3.5.4.3 The Generated Derived Proxy Header

```
#ifndef __demo_chapters_san_inheritance_InheritanceDemoProxy_H
#define __demo_chapters_san_inheritance_InheritanceDemoProxy_H


#include <jni.h>
#include <string>

#include "demo\chapters\san\inheritance\InheritanceDemoBaseProxy.h"
```

```
namespace demo
{
        namespace chapters
        {
                namespace san
                {
                        namespace inheritance
                        {
1:
                                class InheritanceDemoProxy : public
demo::chapters::san::inheritance::InheritanceDemoBaseProxy

                                {
                                private:
                                        static std::string className;
                                        static jclass objectClass;
                                        jobject peerObject;

                                protected:
                                        InheritanceDemoProxy(void* unused);
                                        virtual jobject _getPeerObject() const;

                                public:
                                        static jclass _getObjectClass();
                                        static inline std::string _getClassName()
                                        {
                                                return className;
                                        }

                                        jclass getObjectClass();
                                        operator jobject();
                                        // constructors
                                        InheritanceDemoProxy(jobject obj);
                                        InheritanceDemoProxy();

                                        virtual ~InheritanceDemoProxy();
                                        InheritanceDemoProxy& operator=(const InheritanceDemoProxy&
rhs);

                                        // methods
                                        /*
                                         * void printMessage();
                                         */
                                        void printMessage();

                                };
                        }
                }
        }
}

#endif
```

Notes:

1. This class publicly derives from the generated `InheritanceDemoBaseProxy` class just as illustrated in the class diagram.

Code Snip 3 - 24: The Derived Proxy Header

3.5.4.4 The Generated Derived Proxy Implementation

```
#include "net/sourceforge/jnipp/JNIEnvHelper.h"
#include "InheritanceDemoProxy.h"

using namespace net::sourceforge::jnipp;
using namespace demo::chapters::san::inheritance;


std::string InheritanceDemoProxy::className = "demo/chapters/san/inheritance/InheritanceDemo";
jclass InheritanceDemoProxy::objectClass = NULL;

jclass InheritanceDemoProxy::_getObjectClass()
{
        if ( objectClass == NULL )
```

```
                objectClass = static_cast<jclass>( JNIEnvHelper::NewGlobalRef( JNIEn-
vHelper::FindClass( className.c_str() ) ) );

        return objectClass;
}

1:

InheritanceDemoProxy::InheritanceDemoProxy(void* unused)
        : demo::chapters::san::inheritance::InheritanceDemoBaseProxy( unused )


{
}

jobject InheritanceDemoProxy::_getPeerObject() const
{
        return peerObject;
}

jclass InheritanceDemoProxy::getObjectClass()
{
        return _getObjectClass();
}

InheritanceDemoProxy::operator jobject()
{
        return _getPeerObject();
}

// constructors
InheritanceDemoProxy::InheritanceDemoProxy(jobject obj)
        : demo::chapters::san::inheritance::InheritanceDemoBaseProxy( reinterpret_cast<void*>(NULL) )

{
        peerObject = JNIEnvHelper::NewGlobalRef( obj );
}

InheritanceDemoProxy::InheritanceDemoProxy()
        : demo::chapters::san::inheritance::InheritanceDemoBaseProxy( reinterpret_cast<void*>(NULL) )

{
        jmethodID mid = JNIEnvHelper::GetMethodID( _getObjectClass(), "<init>", "()V" );
        peerObject = JNIEnvHelper::NewGlobalRef( JNIEnvHelper::NewObject( _getObjectClass(), mid ) );
}

InheritanceDemoProxy::~InheritanceDemoProxy()
{
        JNIEnvHelper::DeleteGlobalRef( peerObject );
}

InheritanceDemoProxy& InheritanceDemoProxy::operator=(const InheritanceDemoProxy& rhs)
{
        JNIEnvHelper::DeleteGlobalRef( peerObject );
        peerObject = JNIEnvHelper::NewGlobalRef( rhs.peerObject );
        return *this;
}

// methods
void InheritanceDemoProxy::printMessage()
{
        static jmethodID mid = NULL;
        if ( mid == NULL )
                mid = JNIEnvHelper::GetMethodID( _getObjectClass(), "printMessage", "()V" );
        JNIEnvHelper::CallVoidMethod( _getPeerObject(), mid );
}
```

Notes:

1. Here is that silly constructor call again. This time, we are calling the `InheritanceDemoBaseProxy` constructor.

Code Snip 3 - 25: The Derived Proxy Implementation

At this point, you may be questioning the need for the `void*` constructor that we've been poking fun at. As it turns out, it is necessary. It is used where one would normally use a default constructor. The reason we cannot use a default constructor is that default constructors are often generated by the Proxy generator, and these generated default constructors attempt to create an instance of the associated Java class. This is, of course, an undesirable side effect when the real intent is simply to initialize the base class. Hence the need for our amusing little `void*` constructor.

3.5.5 Using the Generated Proxy Classes

Once again, the supplied driver code locates and loads the JVM before exercising the generated proxy classes. The annotated source is shown below.

```
#include <iostream.h>
#include "net/sourceforge/jnipp/JVM.h"
#include "net/sourceforge/jnipp/JNIEnvHelper.h"
#include "demo/chapters/san/inheritance/InheritanceDemoProxy.h"
#include "net/sourceforge/jnipp/BaseException.h"
#include "net/sourceforge/jnipp/JStringHelper.h"

using namespace net::sourceforge::jnipp;
using namespace demo::chapters::san::inheritance;

int main(int argc, char** argv)
{
        try
        {
                /*
                 * Note that we are relying on the JVM class to resolve the location of the JVM
                 * through the JVM_HOME environment variable.  An exception will be thrown if it
                 * is not set.
                 */
                JVM::load();

1:
                InheritanceDemoProxy derivedInstance;
                derivedInstance.printMessage();

2:
                // the printBaseMessage() is inherited
                derivedInstance.printBaseMessage();

3:
                // the toString() method is inherited also
                cerr << JStringHelper( derivedInstance.toString() ) << endl;

4:
                InheritanceDemoBaseProxy idbp( derivedInstance );
                derivedInstance.printMessage();

5:
                InheritanceDemoBaseProxy baseInstance;
                baseInstance.printMessage();

                JVM::unload();
        }
        catch(BaseException& ex)
        {
                cerr << "caught exception: " << ex.getMessage().c_str() << endl;
        }

        return 0;
}
```

Notes:

1. No surprises here. We simply create an instance of the `InheritanceDemoProxy` class and call one of its methods.
2. Okay, this is new. Although we've created an instance of the derived class, we can call its inherited methods which are passed across the JNI to the base class.
3. Methods of any parent all the way up the tree can be called. Here we call the `toString()` method of the `ObjectProxy` class.
4. At first glance we might expect the `printMessage()` method of the base class to be called. Recall, however, that *all* methods are virtual in Java, and this call behaves exactly as the same code written in Java. The method call is bound to the class of the instance, not the

class of the variable. We therefore are calling the derived class method.

5. Because we have created an instance of the base class, this method *is* bound to the base class and we get the expected result.

Code Snip 3 - 26: The Driver Code

3.5.6 Building and Running

Again, the "native" target is specified for the Ant build file to build the executables. The output for the build is omitted, but the results of the sample run are shown below.

```
C:\development\jnipp\demo\chapters\san>java org.apache.tools.ant.Main -buildfile inheritance.xml run
Buildfile: inheritance.xml

run:
     [exec] Hello from InheritanceDemo
     [exec] Hello from InheritanceDemoBase
     [exec] demo.chapters.san.inheritance.InheritanceDemo@253498
     [exec] Hello from InheritanceDemo
     [exec] Hello from InheritanceDemoBase

BUILD SUCCESSFUL

Total time: 1 second

C:\development\jnipp\demo\chapters\san>
```

Output Sample 3 - 9: Results of Sample Run

3.5.7 An Alternative to Inheritance

Although the "useInheritance" setting is convenient and kinda cool, it is not absoulutely necessary. Similar results could be achieved by foregoing the "useInheritance" settinga and simply adding the superclass to the list of input classes. Because each and every generated proxy class has a constructor that accepts a `jobject` as parameter as well as a `jobject()` typecast operator, it is possible to achieve polymorphic behavior without inheritance. For instance, the project file for the previous example could be set up as shown:

```
<?xml version="1.0"?>

<project name="inheritance" targetType="exe" targetName="inheritanceDemo">
        <proxygen useRichTypes="false">
                <input-classes>
                        <input-class name="demo.chapters.san.inheritance.InheritanceDemo"/>
1:
                        <input-class name="demo.chapters.san.inheritance.InheritanceDemo"/>
                        <input-class name="demo.chapters.san.inheritance.InheritanceDemoBase"/>

                </input-classes>
        </proxygen>
        <nmakefile name="inheritance.mak"/>
        <gnumakefile name="Makefile"/>
</project>
```

Notes:

1. Rather than specifying the use of inheritance, this project file achieves similar results by specifying the superclass as part of the `input-classes`.

Code Snip 3 - 27: Alternative Project File

Specifying the superclass as input classes will result in generated proxy class for both the derived and base classes. It then becomes possible to use the generated proxy for the superclass when a base method needs to be called:

```
                InheritanceDemoProxy derivedProxy;
                InheritanceDemoBaseProxy baseProxy( derivedProxy );
                baseProxy.printBaseMessage();
```

Code Snip 3 - 28: Achieving Polymorphic Behavior without Inheritance

Note, however, that without specifying inheritance, the following is no longer possible:

```
                InheritanceDemoProxy derivedProxy;
                derivedProxy.printBaseMessage();        // InheritanceDemoProxy::printBaseMessage()
does not exist!
```

Code Snip 3 - 29: Limitations when not Using Inheritance

Although it is somewhat less convenient to use code generated without inheritance, the use of this setting can result in numerous generated classes that are never used. The advantage of using the alternative approach is that you can pick and choose which proxy classes are generated. We could have, for example, specified `DemoInterfaceOne` as an input class but not `DemoInterfaceTwo`. This would result in a proxy class that enables the methods of `DemoInterfaceOne` to be called. Perhaps we need to call the methods of this interface, but not those of `DemoInterfaceTwo`. With the "useInheritance" option set, proxy classes would be generated for both, regardless of whether they are used.

3.6 Recursion

One of the more powerful features of the JNI++ Proxy Generator is its ability to automatically generate C++ proxy classes for all of the field, parameter and return types of the input Java classes, *recursively*. This provides the capability of generating C++ proxy classes for *all* of the Java classes that you will likely need to effectively use the target Java class.

3.6.1 The Input Java Class

To illustrate the recursive code generation capabilities of JNI++, we will recycle the example used to illustrate the method name disambiguation feature. If you recall, this example defined a Java class with two `printValue()` methods, one accepting a `java.lang.Integer` parameter and the other a `java.lang.Double`. The Java class is shown below.

```
package demo.chapters.san.recursion;

public class RecursionDemo
{
        public void printValue(Integer value)
        {
                System.out.println( "value is " + value );
        }

        public void printValue(Double value)
        {
                System.out.println( "value is " + value );
        }
}
```

Code Snip 3 - 30: The Input Java Class

3.6.2 The Project File

Looking back a few sections at the original example reveals that we specified two additional classes in the `input-classes` section -- one for each parameter type. Notice that we do not include these in the project file below. Instead, we will opt for the use of recursive code generation.

```
   <?xml version="1.0"?>

   <project name="recursion" targetType="exe" targetName="recursionDemo">

1:
          <proxygen useRichTypes="false" recursionLevel="1">

                  <input-classes>
                        <input-class name="demo.chapters.san.recursion.RecursionDemo"/>
                  </input-classes>
          </proxygen>
          <nmakefile name="recursion.mak"/>
          <gnumakefile name="Makefile"/>
   </project>
```

Notes:

1. The presence of the `recursionLevel` attribute signals the use of recursive code generation. This setting tells the code generator to generate proxy classes for all field, parameter and return types used in the input Java class, but no deeper.

Code Snip 3 - 31: The Project File

3.6.3 Generating the Code

Using the supplied Ant build file, the code is generated in the usual manner by specifying the "generate" target upon invocation. This time, however, C++ proxy classes are *automatically* generated for both the `java.lang.Integer` and `java.lang.Double` parameter types. The specified `recursionLevel` directs the code generator to generate proxy classes for the field, parameter and return types only of the input `demo.chapters.san.recursion.RecursionDemo` Java class. Specifying a `recursionLevel` of 2 would cause the proxy classes to also be generated for the `java.lang.Integer` and `java.lang.Double` classes.

```
C:\development\jnipp\dist\demo\chapters\san>java org.apache.tools.ant.Main -buildfile recursion.xml
generate
Buildfile: recursion.xml

prepare:

compile:
    [javac] Compiling 1 source file to C:\development\jnipp\dist\build\classes

generate:
     [java] generating C++ Proxy Class for demo.chapters.san.recursion.Recursion
Demo ...
     [java] generating C++ Proxy Class for java.lang.Double ...
     [java] generating C++ Proxy Class for java.lang.Integer ...
     [java] generating NMakefile for project recursion ...
     [java] generating GNU Makefile for project recursion ...
     [java]
     [java] code generation complete.

BUILD SUCCESSFUL

Total time: 7 seconds
```

```
C:\development\jnipp\dist\demo\chapters\san>
```

Output Sample 3 - 10: Generating the Recursive Code

3.6.4 Using the Generated Code

```
#include <iostream.h>
#include "net/sourceforge/jnipp/JVM.h"
#include "net/sourceforge/jnipp/JNIEnvHelper.h"
#include "demo/chapters/san/recursion/RecursionDemoProxy.h"
#include "java/lang/IntegerProxy.h"
#include "java/lang/DoubleProxy.h"
#include "net/sourceforge/jnipp/BaseException.h"

using namespace net::sourceforge::jnipp;
using namespace demo::chapters::san::recursion;
using namespace java::lang;

int main(int argc, char** argv)
{
        try
        {
                /*
                 * Note that we are relying on the JVM class to resolve the location of the JVM
                 * through the JVM_HOME environment variable.  An exception will be thrown if it
                 * is not set.
                 */
                JVM::load();

                RecursionDemoProxy rdp;
                rdp.printValue( DoubleProxy( 123.456 ) );
                rdp.printValue1( IntegerProxy( 123 ) );

                JVM::unload();
        }
        catch(BaseException& ex)
        {
                cerr << "caught exception: " << ex.getMessage().c_str() << endl;
        }

        return 0;
}
```

Code Snip 3 - 32: The Driver Code

3.6.5 Building and Running

Using the same buildfile and specifying the "native" target, the debug and release modules are built. As in the previous example, the output for this and subsequent examples will omit the output to save space.

With the newly-built executable, the test run can be invoked by specifying the "run" target using the supplied Ant build file. The output, shown below, shows no surprises.

```
C:\development\jnipp\demo\chapters\san>java org.apache.tools.ant.Main -buildfile recursion.xml run
Buildfile: recursion.xml

run:
     [exec] value is 123.456
     [exec] value is 123

BUILD SUCCESSFUL

Total time: 1 second
```

```
C:\development\jnipp\demo\chapters\san>
```

Output Sample 3 - 11: Running the Example

3.7 Rich Data Types

The recursive capability of the JNI++ Proxy generator is certainly a powerful tool -- it generates most, if not all, of the C++ proxy classes that will be needed for the input Java classes. The generated code, however, still utilizes raw JNI data types for the field, parameter and return types. This is not optimal for a number of reasons. First, for reasons discussed in the first section of this chapter, the method names must often times be disambiguated, leading to method names that differ from those of their Java counterparts. If two constructors lead to the same JNI signature then the code won't even compile. Second, it is not obvious from looking at the generated C++ method exactly what types it expects as parameters and what type it returns -- the raw JNI data types are too generic. One must be familiar with (or constantly reference) the associated Java class signatures. Third, compile-time type checking is also weak -- practically *anything* can be passed as a `jobject`, for example. For these reasons and possibly others, the JNI++ Proxy Generator supports the use of rich data type generation, which extends the recursive capability by also generating proxy and helper classes as field, parameter and return types.

3.7.1 The Project File

The project file for this example is shown below. Note that all of the examples up to this point specified a `useRichTypes` attribute of "false". The default is "true", so it is simply omitted here. Also note the new `usePartialSpec` attribute of the `project` element. This is used to specify an alternate means of generating code for use of the `::net::sourceforge::jnipp::ProxyArray` template helper class for those compilers that do not support partial template specialization. MS Visual C++ does not support partial template specialization, so we set the attribute to "false". The default is "true", and GNU g++ users should simply omit the attribute to utilize the default setting. We will see more of the `ProxyArray` class in the following chapter.

```
   <?xml version="1.0"?>
1:
   <project name="rich" targetType="exe" targetName="richDemo" usePartialSpec="false">

           <proxygen>
                   <input-classes>
                           <input-class name="demo.chapters.san.rich.RichDemo"/>
                   </input-classes>
           </proxygen>
           <nmakefile name="rich.mak"/>
           <gnumakefile name="Makefile"/>
   </project>
```

Notes:

1. Set `usePartialSpec` to "false" if your compiler does not support partial template specialization (MS VC++, for one). If you are using GNU g++, omit the `usePartialSpec` attribute.

Code Snip 3 - 33: The Project File

3.7.2 Generating the Code

Invoking the Ant build tool with the `rich.xml` project file as input and specifying the "generate" target produces numerous files. You may be surprised at the number of files generated and wonder whether they are all neces-

sary. The `recursionLevel` attribute we saw earlier has no effect when using "rich types", although support for this is planned for a future release. The result is total recursive code generation, and the number of files generated is equivalent to the number generated if `useRichTypes` is "false" and the `recursionLevel` is set to `0`. C++ proxy classes are generated for *all* field, parameter and return types, recursively -- and not to a specified depth. This ensures that all methods and constructors are unique and is the reason that the method name disambiguation feature is unnecessary when using "rich types". For obvious reasons, the output of the code generation step has been omitted.

3.7.3 The Generated Proxy Class

Although a large number of files have been generated, we will take a quick look only at the generated proxy class for our input `demo.chapters.san.rich.RichDemo` Java class. The generated code for the header and implementation is shown below.

```
#ifndef __demo_chapters_san_rich_RichDemoProxy_H
#define __demo_chapters_san_rich_RichDemoProxy_H


#include <jni.h>
#include <string>
1:

#include "net/sourceforge/jnipp/JBooleanArrayHelper.h"
#include "net/sourceforge/jnipp/JByteArrayHelper.h"
#include "net/sourceforge/jnipp/JCharArrayHelper.h"
#include "net/sourceforge/jnipp/JDoubleArrayHelper.h"
#include "net/sourceforge/jnipp/JFloatArrayHelper.h"
#include "net/sourceforge/jnipp/JIntArrayHelper.h"
#include "net/sourceforge/jnipp/JLongArrayHelper.h"
#include "net/sourceforge/jnipp/JShortArrayHelper.h"
#include "net/sourceforge/jnipp/JStringHelper.h"
#include "net/sourceforge/jnipp/JStringHelperArray.h"
#include "net/sourceforge/jnipp/ProxyArray.h"


2:

// includes for parameter and return type proxy classes
#include "java\util\CollectionProxyForward.h"


namespace demo
{
        namespace chapters
        {
                namespace san
                {
                        namespace rich
                        {
                                class RichDemoProxy
                                {
                                private:
                                        static std::string className;
                                        static jclass objectClass;
                                        jobject peerObject;

                                protected:
                                        RichDemoProxy(void* unused);
                                        virtual jobject _getPeerObject() const;

                                public:
                                        static jclass _getObjectClass();
                                        static inline std::string _getClassName()
                                        {
                                                return className;
                                        }

                                        jclass getObjectClass();
                                        operator jobject();
                                        // constructors
                                        RichDemoProxy(jobject obj);
                                        RichDemoProxy();

                                        virtual ~RichDemoProxy();
```

```
                                        RichDemoProxy& operator=(const RichDemoProxy& rhs);

                                        // methods

3:
                                        /*
                                         * void printCollection(Collection);
                                         */
                                        void printCollection(::java::util::CollectionProxy p0);

                                };
                        }
                }
        }
}

#endif
```

Notes:

1. These `#includes` are generated for all proxy header files when using "rich types". These are all of the JNI++ "array helper" classes.
2. As mentioned earlier in the first example of the chapter, the "forward" header is used in the `include` directives in the generated proxy header files. The "forward" header files are used to avoid recursion in the `include` files.
3. Here is our `printCollection` proxy method. Note the parameter type is itself a generated proxy class rather than a `jobject`.

Code Snip 3 - 34: Generated Proxy Class Header with Rich Types

The generated proxy implementation is very similar to that generated without "rich types", with a few exceptions dealing with the proxy field, parameter and return types. The code is shown below.

```
#include "net/sourceforge/jnipp/JNIEnvHelper.h"
#include "RichDemoProxy.h"

1:
// includes for parameter and return type proxy classes
#include "java\util\CollectionProxy.h"


using namespace net::sourceforge::jnipp;
using namespace demo::chapters::san::rich;


std::string RichDemoProxy::className = "demo/chapters/san/rich/RichDemo";
jclass RichDemoProxy::objectClass = NULL;

jclass RichDemoProxy::_getObjectClass()
{
        if ( objectClass == NULL )
                objectClass = static_cast<jclass>( JNIEnvHelper::NewGlobalRef( JNIEn-
vHelper::FindClass( className.c_str() ) ) );

        return objectClass;
}

RichDemoProxy::RichDemoProxy(void* unused)
{
}

jobject RichDemoProxy::_getPeerObject() const
{
        return peerObject;
}

jclass RichDemoProxy::getObjectClass()
{
        return _getObjectClass();
}

RichDemoProxy::operator jobject()
{
        return _getPeerObject();
}

// constructors
```

```
RichDemoProxy::RichDemoProxy(jobject obj)
{
        peerObject = JNIEnvHelper::NewGlobalRef( obj );
}

RichDemoProxy::RichDemoProxy()
{
        jmethodID mid = JNIEnvHelper::GetMethodID( _getObjectClass(), "<init>", "()V" );
        peerObject = JNIEnvHelper::NewGlobalRef( JNIEnvHelper::NewObject( _getObjectClass(), mid ) );
}

RichDemoProxy::~RichDemoProxy()
{
        JNIEnvHelper::DeleteGlobalRef( peerObject );
}

RichDemoProxy& RichDemoProxy::operator=(const RichDemoProxy& rhs)
{
        JNIEnvHelper::DeleteGlobalRef( peerObject );
        peerObject = JNIEnvHelper::NewGlobalRef( rhs.peerObject );
        return *this;
}
```

```
2:

// methods
void RichDemoProxy::printCollection(::java::util::CollectionProxy p0)
{
        static jmethodID mid = NULL;
        if ( mid == NULL )
                mid = JNIEnvHelper::GetMethodID( _getObjectClass(), "printCollection",
 "(Ljava/util/Collection;)V" );
        JNIEnvHelper::CallVoidMethod( _getPeerObject(), mid, static_cast<jobject>( p0 ) );
}
```

Notes:

1. Here is the include for the `CollectionProxy` parameter type that contains the class definition.
2. The generated `printCollection()` method accepts a `CollectionProxy` instance. Note the use of `operator jobject()` when the parameter is passed into `CallVoidMethod()`.

Code Snip 3 - 35: Generated Proxy Class Implementation with Rich Types

3.7.4 Exercising the Proxy Class

The `main()` driver method is shown below. Note the use of the `ArrayListProxy`, `CollectionProxy` and `ObjectProxy` classes. These were all generated as a result of the recursive nature of the rich type. The `ArrayListProxy::add()` method accepts an `ObjectProxy` as parameter, so we must wrap the `JStringHelper` in an `ObjectProxy` instance. Note also the use of the "inheritance alternative" introduced earlier. We can instantiate a `CollectionProxy` instance using the `ArrayListProxy` instance because the `ArrayList` class *implements* `Collection` in the input Java source.

```
#include <iostream.h>
#include "net/sourceforge/jnipp/JVM.h"
#include "net/sourceforge/jnipp/JNIEnvHelper.h"
#include "demo/chapters/san/rich/RichDemoProxy.h"
#include "java/util/ArrayListProxy.h"
#include "java/util/CollectionProxy.h"
#include "java/lang/ObjectProxy.h"
#include "net/sourceforge/jnipp/BaseException.h"

using namespace net::sourceforge::jnipp;
using namespace demo::chapters::san::rich;
using namespace java::util;
using namespace java::lang;

int main(int argc, char** argv)
{
        try
        {
                /*
                 * Note that we are relying on the JVM class to resolve the location of the JVM
```

```
                    * through the JVM_HOME environment variable.  An exception will be thrown if it
                    * is not set.
                    */
                   JVM::load();

                   RichDemoProxy rdp;
                   ArrayListProxy alp;

1:

                   alp.add( ObjectProxy( JStringHelper( "Come" ) ) );
                   alp.add( ObjectProxy( JStringHelper( "up" ) ) );
                   alp.add( ObjectProxy( JStringHelper( "and" ) ) );
                   alp.add( ObjectProxy( JStringHelper( "C++" ) ) );
                   alp.add( ObjectProxy( JStringHelper( "me" ) ) );
                   alp.add( ObjectProxy( JStringHelper( "sometime" ) ) );

                   // note the use of the "inheritance alternative" -- instantiating a CollectionProxy
  with the
                   // ArrayListProxy (rather than ArrayListProxy derived from CollectionProxy)
                   rdp.printCollection( CollectionProxy( alp ) );
                   JVM::unload();
           }
           catch(BaseException& ex)
           {
                   cerr << "caught exception: " << ex.getMessage().c_str() << endl;
           }

           return 0;
   }
```

Notes:

1. The `JStringHelper`'s `operator jobject()` is used to create an instance of the generated `ObjectProxy` class, which is then `add()`ed to the `ArrayList`.

Code Snip 3 - 36: The Driver Code

3.7.5 Building and Running

Using the same buildfile and specifying the "native" target, the debug and release modules are built. As in the previous example, the output for this example will omit the output to save space.

With the newly-built executable, the test run can be invoked by specifying the "run" target using the supplied Ant build file. The output, shown below, shows no surprises.

```
C:\development\jnipp\demo\chapters\san>java org.apache.tools.ant.Main -buildfile rich.xml run
Buildfile: rich.xml

run:
     [exec] Come
     [exec] up
     [exec] and
     [exec] C++
     [exec] me
     [exec] sometime

BUILD SUCCESSFUL

Total time: 1 second

C:\development\jnipp\demo\chapters\san>
```

Output Sample 3 - 12: Running the Example

3.8 Inner Classes

3.9 Option Summary

We have covered a lot in this chapter -- hopefully the table below will help summarize. Each of the options for the JNI++ Proxy Generator is listed along with its default value.

Attribute Name	Required?	Default	Description
attributeGetters	No	false	If "true", directs JNI++ to generate attribute accessor methods for all declared fields for all of the generated proxy classes.
attributeSetters	No	false	If "true", directs JNI++ to generate attribute mutator methods for all declared fields for all of the generated proxy classes.
useInheritance	No	false	If "true", directs JNI++ to generate inheritance hierarchies in the proxy classes-that mimic the Java inputs.
recursionLevel	No	0	If non-negative, directs JNI++ to generate proxy classes for all field, parameter and return types referenced in the input classes, recursively, up to the level specified. If specified as zero, then all proxy classes required to satisfy the field, parameter and return types of all of the inputs are generated.
useRichTypes	No	true	If true, JNI++ will not only generate recursively as above, but will also generate the field, parameter and return types using proxy and helper classes.
innerClasses	No	false	If true, directs JNI++ to generate proxy classes for all inner classes of the input Java classes.

Chapter 4

The JNI++ Peer Generator

4.1 Overview

The JNI++ Peer Generator completes the toolset by generating C++ and Java source that enables the use of C++ classes as if they were implemented in Java. This utility alone is a convenience, but when combined with the JNI++ Proxy Generator it becomes a very powerful tool. With the two combined tools, code can not only be generated to simplify writing the C++ service, but also to access the entire Java API subset using rich data types (helper classes and generated C++ proxy classes). This chapter will illustrate the features of the JNI++ Peer Generator in a similar manner as the previous chapter. Each of the annotated examples will step through the use of a particular setting or combination of settings. Again, the source for all of the examples is located in the `JNIPP_HOME/demo/chapters` directory of the distribution.

The JNI++ Peer Generator accepts a project file and one or more Java interfaces as input. It then takes the input Java interfaces and generates code necessary to effortlessly implement each in C++. It can also optionally generate proxy classes for all of the return and parameter types and makefiles for the target platforms to further simplify your work.

The first step is to create a project file that lists the input Java interfaces and sets various code generation options. Next, the input Java interfaces are compiled. Once again, this second step is required because the JNI++ code generators do not parse the Java source files, but rather rely on Java introspection to query the various metadata for a given input class. Next, the JNI++ Peer Generator is invoked to generate C++ peer classes for the input interfaces. After filling in the implementation of the generated C++ peer classes, they are compiled and linked with user-written code and the core JNI++ untime library to produce either an executable image or shared library.

All of the examples in this chapter follow this exact set of steps to build shared library with simple Java driver code to exercise the generated peer classes. You may find it helpful to refer to this diagram when stepping through the examples.

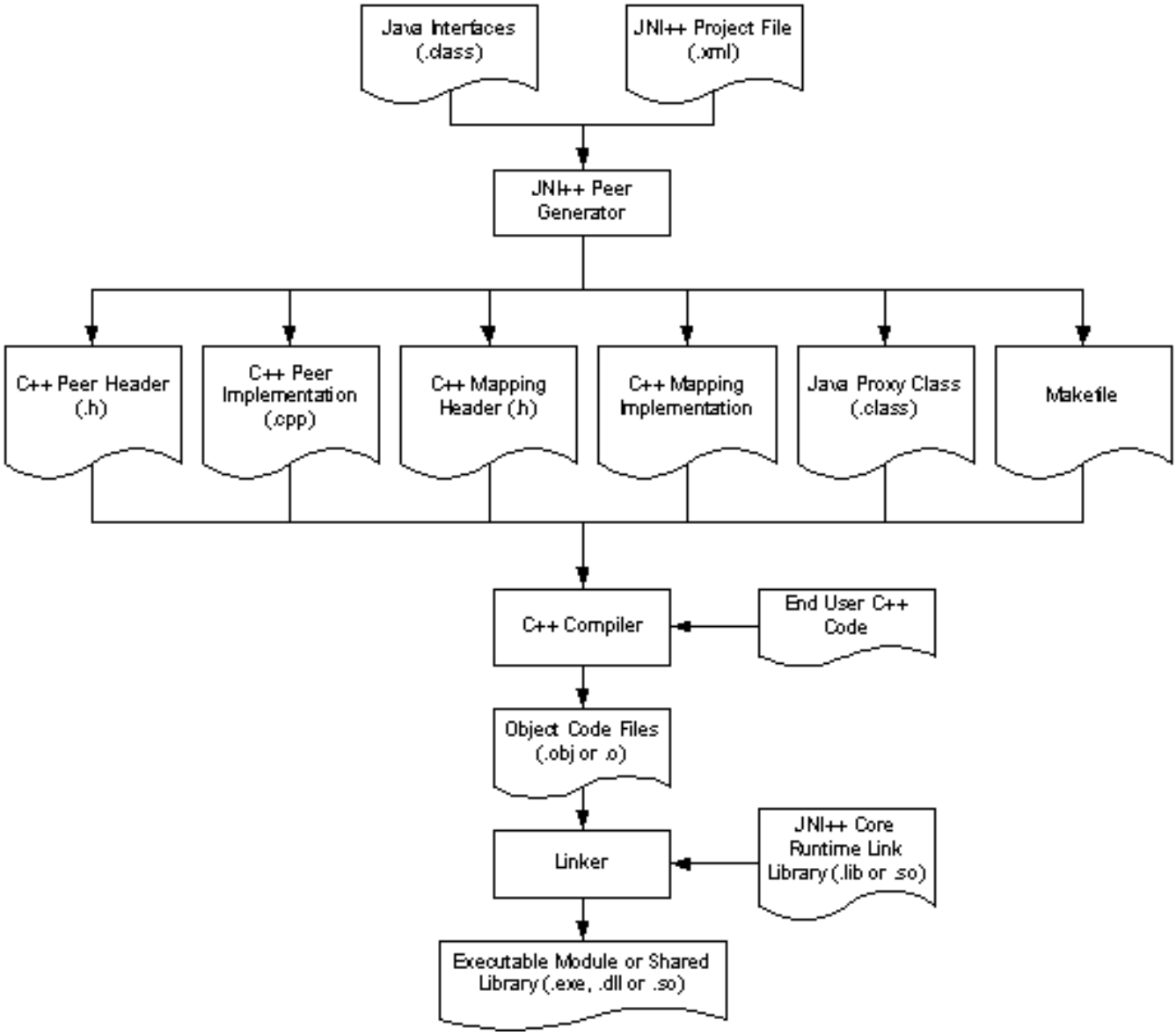

Figure 4 - 1: JNI++ Peer Generation

The following figure illustrates a typical runtime deployment environment for an application that is using generated C++ peer classes. The client Java code running inside the Java Virtual Machine invokes methods on the generated Java proxy class. This generated class implements the interface that was used to generate the peer by defining `native` methods for each. A `loadLibrary()` call in the static initializer ensures the shared library that contains the C++ peer implementation is loaded. Calls to the `native` methods of the Java proxy class are passed by the JVM across to the generated mapping code on the C++ side of the JNI. The procedural mapping code, in turn, marshals the call to the C++ peer instance associated with the Java proxy class. Both the mapping code and the C++ peer instance may make use of the core runtime library to perform their tasks.

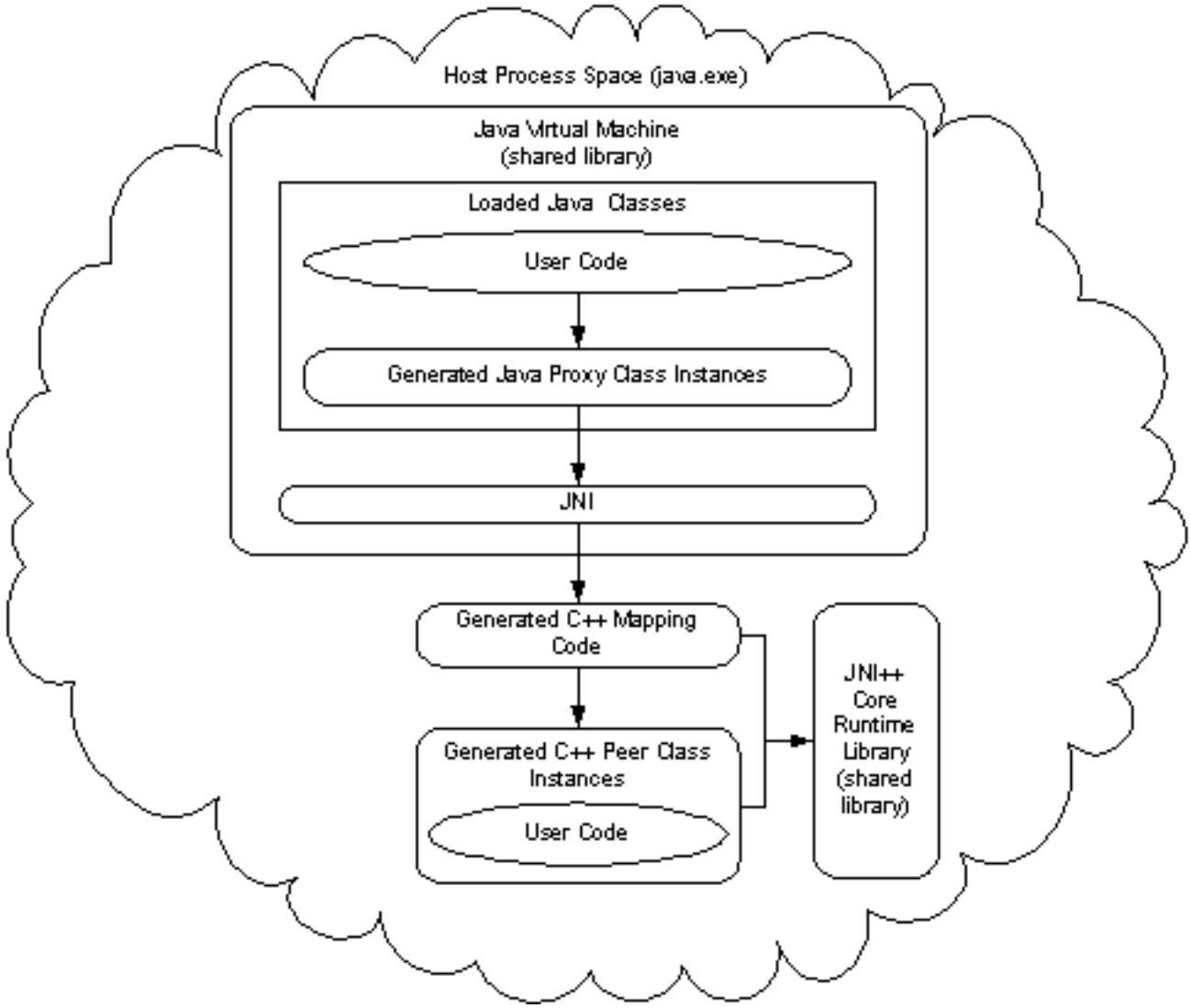

Figure 4 - 2: Typical JNI++ Peer Execution Environment

4.2 Another Very Simple Example

As in the previous chapter illustrating the features of the JNI++ Proxy Generator, this chapter will also begin by walking through a very simple example. Our input Java interface defines a single method, `printMessage()`, that accepts a `java.lang.String` message. The JNI++ Peer Generator accepts only Java interfaces as input, and it is the input Java interface that defines the services that will be implemented in the C++ peer. In its most basic configuration, it will generate the files necessary to implement the service in C++ -- all you need to do is simply fill in the code. Let's see how it all works by stepping through the example.

4.2.1 The Input Java Interface

```
   package demo.chapters.shi.simple;

   public interface SimpleDemo
   {
1:
```

```
        public void printMessage(String message);

}
```

Notes:

1. Here is the single method to be implemented in the C++ peer class. Just as in Java, the input Java interface defines the contract that the C++ peer must implement.

Code Snip 4 - 1: The Input Java Interface

As mentioned in the first example of the previous chapter, the input Java sources must be compiled before the code generators can be invoked. This and the following examples assume that the Java classes are compiled into the `%JNIPP_HOME%\build\classes` directory. The following output sample shows the Java interface compiled in a Win32 environment.

```
C:\development\jnipp\demo\chapters\shi>javac -d %JNIPP_HOME%\build\classes simple\SimpleDemo.java
C:\development\jnipp\demo\chapters\shi>
```

Output Sample 4 - 1: Compiling the Java Interface

4.2.2 The Project File

```
<?xml version="1.0"?>
1:
<project name="simple" targetType="shlib" targetName="SimpleDemo">

        <peergen>
                <input-classes>
                        <input-class name="demo.chapters.shi.simple.SimpleDemo"/>
                </input-classes>
        </peergen>
        <nmakefile name="simple.mak"/>
        <gnumakefile name="Makefile"/>
</project>
```

Notes:

1. For all of the examples in this chapter, we will be creating a shared library that contains our peer implementations. This is achieved by specifying a `targetType` of "shlib".

Code Snip 4 - 2: The Project File

As you can see, this project file leaves all of the Peer Generator settings at their default values and specifies the single Java interface as input. All of the Peer examples will generate a shared library, however, one could just as easily build an executable.

4.2.3 Invoking the Code Generator

As mentioned in the previous chapter, although the GUI can be used to create the project files and invoke the code generators, the examples in this guide will show the invocation of the command line tool. Each of the exam-

ples in the distribution has an associated Ant build file that can be used to simplify the build process. For this example, however, the steps required to build and run the application manually will be illustrated. Subsequent examples will utilize the Ant build file to build and run.

The `jnipp` executable is invoked with a single command line parameter, *-projectFile* , to specify the project file to use for the code generation settings. The `CLASSPATH` must contain not only the the `xerces.jar` file, but also the classes for which you are generating code. The `JVM_HOME` and `JNIPP_HOME` environment variables must also be set according to the guidelines listed in Chapter 2.

```
C:\development\jnipp\demo\chapters\san\simple>set JNIPP_HOME=c:\development\jnipp\dist

C:\development\jnipp\demo\chapters\san\simple>set JVM_HOME=c:\java\jdk1.3\jre\bin\hotspot\jvm.dll

C:\development\jnipp\demo\chapters\san\simple>set
CLASSPATH=c:\jars\xerces.jar;%JNIPP_HOME%\build\classes

C:\development\jnipp\demo\chapters\san\simple>jnipp -projectFile project.xml
generating C++ Peer Class for demo.chapters.shi.simple.SimpleDemo ...
generating NMakefile for project simple ...
generating GNU Makefile for project simple ...

code generation complete.

C:\development\jnipp\demo\chapters\shi\simple>
```

Output Sample 4 - 2: Invoking the JNI++ Peer Generator

4.2.4 A Look at the Generated Files

A total of six source files are generated for each input Java interface. Additionally, makefiles are generated for the entire project. This section will provide a quick review of the generated files. You may find it helpful to refer to the diagram in the *Overview* as you look over the files.

4.2.4.1 The Generated C++ Peer Header

```
#ifndef __demo_chapters_shi_simple_SimpleDemoPeer_H
#define __demo_chapters_shi_simple_SimpleDemoPeer_H


#include <jni.h>
#include <string>

namespace demo
{
        namespace chapters
        {
                namespace shi
                {
                        namespace simple
                        {
                                class SimpleDemoPeer
                                {
                                private:

                                public:
                                        SimpleDemoPeer();

1:
                                        // methods
                                        void printMessage(JNIEnv* env, jobject obj, jstring p0);
```

```
                                    };
                            };
                    };
            };
    };

    #endif
```

Notes:

1. Here is the generated method definition corresponding to the analagous method of the input Java interface.

Code Snip 4 - 3: The C++ Peer Header

4.2.4.2 The Generated C++ Peer Implementation

```
    #include "SimpleDemoPeer.h"

    using namespace demo::chapters::shi::simple;

    SimpleDemoPeer::SimpleDemoPeer()
    {
    }
1:

    // methods
    void SimpleDemoPeer::printMessage(JNIEnv* env, jobject obj, jstring p0)
    {
            // TODO: Fill in your implementation here
    }
```

Notes:

1. Here is the generated method skeleton whose implementation is to be provided. This method is ultimately called when the associated Java proxy method is called.

Code Snip 4 - 4: The C++ Peer Implementation

4.2.4.3 The Generated C++ Mapping Header

```
    #ifndef __demo_chapters_shi_simple_SimpleDemoMapping_H
    #define __demo_chapters_shi_simple_SimpleDemoMapping_H


    #include <jni.h>
    #include "SimpleDemoPeer.h"
1:

    extern "C"
    {
            JNIEXPORT void JNICALL Java_demo_chapters_shi_simple_SimpleDemoProxy_init(JNIEnv*, jclass);
    };


2:

    demo::chapters::shi::simple::SimpleDemoPeer*
    Java_demo_chapters_shi_simple_SimpleDemoProxy_getPeerPtr(JNIEnv*, jobject);
    JNIEXPORT void JNICALL Java_demo_chapters_shi_simple_SimpleDemoProxy_releasePeer(JNIEnv*, jobject);


3:
```

```
// methods
JNIEXPORT void JNICALL Java_demo_chapters_shi_simple_SimpleDemoProxy_printMessage(JNIEnv*, jobject,
jstring);



#endif
```

Notes:

1. This is the only method that is ever exported using the `extern "C"` construct. The generated mapping code uses a `RegisterNatives()` call to register the remaining native methods.
2. These methods are common to all generated peers, although the generated names will vary.
3. This is the procedural JNI representation of our input Java interface method.

Code Snip 4 - 5: The C++ Mapping Header

As we will see in the generated implementation, the mapping code provides the procedural entry point that should be familiar to anyone who has JNI programming experience. It contains a single `extern "C"` method that corresponds to the `init()` method of the generated Java proxy. As shown below, the native implementation of this method registers the remaining native methods with the JVM.

4.2.4.4 The Generated C++ Mapping Implementation

```
#include "net/sourceforge/jnipp/JVM.h"
#include "net/sourceforge/jnipp/JNIEnvHelper.h"
#include "net/sourceforge/jnipp/EnvironmentAlreadyInitializedException.h"
#include "SimpleDemoMapping.h"
#include "SimpleDemoPeerFactory.h"

using namespace net::sourceforge::jnipp;
using namespace demo::chapters::shi::simple;


JNIEXPORT void JNICALL Java_demo_chapters_shi_simple_SimpleDemoProxy_init(JNIEnv* env, jclass cls)
{
        // This method is called by the Java Proxy init() on the other side of the JNI to initialize
the environment
        try
        {
                JNIEnvHelper::init( env );

1:

                JNINativeMethod nativeMethods[] =
                {
                        { "releasePeer", "()V",
(void*)Java_demo_chapters_shi_simple_SimpleDemoProxy_releasePeer },
                        { "printMessage", "(Ljava/lang/String;)V",
(void*)Java_demo_chapters_shi_simple_SimpleDemoProxy_printMessage }
                };

                JNIEnvHelper::RegisterNatives( cls, nativeMethods,
sizeof(nativeMethods)/sizeof(nativeMethods[0]) );


        }
        catch(EnvironmentAlreadyInitializedException&)
        {
        }
}

SimpleDemoPeer* Java_demo_chapters_shi_simple_SimpleDemoProxy_getPeerPtr(JNIEnv* env, jobject obj)
{
        jclass cls = env->GetObjectClass( obj );

2:

        jfieldID fid = env->GetFieldID( cls, "peerPtr", "J" );
        jlong peerPtr = env->GetLongField( obj, fid );


        SimpleDemoPeer* ptr = NULL;
```

```
        if ( peerPtr == 0 )
        {
                ptr = SimpleDemoPeerFactory::newPeer();
                peerPtr = reinterpret_cast<jlong>( ptr );
                env->SetLongField( obj, fid, peerPtr );
        }

        return reinterpret_cast<SimpleDemoPeer*>( peerPtr );
}

JNIEXPORT void JNICALL Java_demo_chapters_shi_simple_SimpleDemoProxy_releasePeer(JNIEnv* env, jobject
obj)
{
        // This method is called by the Java Proxy finalizer() on the other side of the JNI to free
the peer
        jclass cls = env->GetObjectClass( obj );
        jfieldID fid = env->GetFieldID( cls, "peerPtr", "J" );
        jlong peerPtr = env->GetLongField( obj, fid );

3:
        if ( peerPtr != 0 )
                delete reinterpret_cast<SimpleDemoPeer*>( peerPtr );

}

// methods
JNIEXPORT void JNICALL Java_demo_chapters_shi_simple_SimpleDemoProxy_printMessage(JNIEnv* env, job-
ject obj, jstring p0)
{
        JNIEnvHelper::init( env );
        try
        {

4:
                Java_demo_chapters_shi_simple_SimpleDemoProxy_getPeerPtr( env, obj )->printMessage(
env, obj, p0 );

        }
        catch(jthrowable thr)
        {
                JNIEnvHelper::Throw( thr );
        }
}
```

Notes:

1. The native methods corresponding to those of the input Java interface must be registered with the JVM before they are called. This is performed in the `init()` method, called after the library is loaded.
2. The pointer to the C++ peer is stored in the `peerPtr` field of the generated Java proxy class instance.
3. The lifetime of the C++ peer instance is bracketed by the lifetime of the associated Java proxy instance. This method is ultimately called from the `finalize()` method of the Java proxy.
4. The call is ultimately just passed on to the associated C++ peer instance for processing.

Code Snip 4 - 6: The C++ Mapping Implementation

The generated mapping code performs the critical function of bridging the Java and C++ worlds. It provides a conduit through which the native method calls on the generated Java proxy instance travel to the associated C++ peer implementation. Those familiar with JNI programming may be curious as to why the `RegisterNatives()` call is made rather than simply bracketing all of the methods in an `extern "C"` construct like the `init()` method. First, doing so would require the method names to be unnecessarily mangled to conform to the JNI specification. Using the `RegisterNatives()` approach carries no naming restriction. The second reason for this approach is so that the same generated code will work regardless of whether the C++ peers live in a shared library (as they will here and are in typical JNI programming) or within an executable image that launches the JVM. If an executable is generated, there would be no `loadLibrary()` call, and therefore would be no automatic registration of the native methods. We must therefore make the call from our generated code as we see here.

4.2.4.5 The Generated Java Proxy

The JNI++ Peer Generator also generates a Java class that serves as proxy for the generated C++ peer. This class defines a `long` field that contains the pointer for the C++ peer and is used in the generated mapping code (see above). It also defines a native `init()` method that is called immediately after loading the shared library in the `static` block (the `loadLibrary()` call only occurs if the project is targeting a shared library). The `init()` method is covered above in the explanatory text of the generated mapping code. Each method of the input Java interface results in a `native` method in the generated Java proxy class (note that the generated proxy class `implements` the input interface). The real implementation is left to the C++ peer class.

```
   package demo.chapters.shi.simple;

   import demo.chapters.shi.simple.SimpleDemo;

1:

   public class SimpleDemoProxy
           implements SimpleDemo

   {

2:

           private long peerPtr = 0;


3:

           private static native void init();
           private native void releasePeer();

           protected void finalize()
                   throws Throwable
           {
                   releasePeer();
           }

4:

           // methods
           public native void printMessage(java.lang.String p0);


5:

           static
           {
                   System.loadLibrary( "SimpleDemo" );
                   init();
           }

   }
```

Notes:

1. Each input Java interface results in a generated Java proxy class that implements the interface.
2. The pointer to the C++ peer instance associated with this proxy.
3. All generated Java proxy classes define these two methods. The `init()` method is used to register the native methods and the `releasePeer` method is called from the `finalize()` method to free the associated C++ peer instance.
4. Each method of the input Java interface results in a `native` method in the generate Java proxy class.
5. This `static` block is executed once when the class is loaded and takes care of loading the shared library (unless the project is targeting an executable) and calling the `init()` method which registers the remaining native methods.

Code Snip 4 - 7: The Java Proxy

4.2.4.6 The Generated Peer Factory Header

The last generated file is a simple factory that is called by the mapping code to create instances of the C++ peer implementation. As we will see shortly, the `useInheritance` attribute can be set such that the implementation

is provided in a derived class, and we will modify the generated factory to create instances of the derived implementation class. For this example, however, the generated file remains in its unmodified state shown below.

```
#ifndef __demo_chapters_shi_simple_SimpleDemoPeerFactory_H
#define __demo_chapters_shi_simple_SimpleDemoPeerFactory_H


#include "SimpleDemoPeer.h"

namespace demo
{
        namespace chapters
        {
                namespace shi
                {
                        namespace simple
                        {
                                class SimpleDemoPeerFactory
                                {
                                public:
                                        static inline SimpleDemoPeer* newPeer()
                                        {
                                                return new SimpleDemoPeer;
                                        }
                                };
                        };
                };
        };
};


#endif
```

Code Snip 4 - 8: The Peer Factory Header

4.2.4.7 The Generated Makefile

```
!IF "$(CFG)" != "Release" && "$(CFG)" != "Debug"
!MESSAGE You must specify a configuration by defining the macro CFG on the command line. For example:
!MESSAGE
!MESSAGE NMAKE /f simple.mak CFG="Debug"
!MESSAGE
!MESSAGE Possible choices for configuration are:
!MESSAGE
!MESSAGE "Release"
!MESSAGE "Debug"
!MESSAGE
!ERROR An invalid configuration is specified.
!ENDIF

!IF "$(OS)" == "Windows_NT"
NULL=
!ELSE
NULL=nul
!ENDIF

1:

TARGETTYPE=DLL


2:

SOURCES= .\demo\chapters\shi\simple\SimpleDemoPeer.cpp
.\demo\chapters\shi\simple\SimpleDemoMapping.cpp

OBJS=$(SOURCES:.cpp=.obj)

CPP=cl.exe
INC=/I .\ /I "$(JAVA_HOME)/include" /I "$(JAVA_HOME)/include/win32" /I "$(JNIPP_HOME)/include"
CPPFLAGS=/nologo /GX /W3 /FD /D "WIN32" /D "_WINDOWS" /D "_MBCS" $(INC) /c
LINK=link.exe
LINKFLAGS=/nologo /machine:IX86 /libpath:"$(JNIPP_HOME)/lib"
```

```
!IF "$(TARGETTYPE)" == "DLL"
CPPFLAGS=$(CPPFLAGS) /D "_USRDLL"
EXT="dll"
LINKFLAGS=$(LINKFLAGS) /incremental:no /dll
!ELSEIF "$(TARGETTYPE)" == "CONSOLEAPP"
EXT="exe"
LINKFLAGS=$(LINKFLAGS) /subsystem:console /incremental:no
!ENDIF

!IF "$(CFG)" == "Debug"

CPPFLAGS=$(CPPFLAGS) /MDd /Zi /Od /D "_DEBUG"
OUTDIR=Debug
LINKFLAGS=$(LINKFLAGS) /debug
LINKOBJS=$(OBJS:.obj=_d.obj)
LINKLIBS=libJNIPPCore_d.lib

TARGETNAME=SimpleDemo_d

"$(OUTDIR)":
        if not exist "$(OUTDIR)\$(NULL)" mkdir "$(OUTDIR)"

.cpp.obj:
        $(CPP) @<< $(CPPFLAGS) /Fo$(<:.cpp=_d.obj) $<
<<

!ELSE

CPPFLAGS=$(CPPFLAGS) /MD /O2 /D "NDEBUG"
OUTDIR=Release
LINKOBJS=$(OBJS)
LINKLIBS=libJNIPPCore.lib

TARGETNAME=SimpleDemo

"$(OUTDIR)":
        if not exist "$(OUTDIR)\$(NULL)" mkdir "$(OUTDIR)"

.cpp.obj:
        $(CPP) @<< $(CPPFLAGS) /Fo$(<:.cpp=.obj) $<
<<

!ENDIF

"$(OUTDIR)\$(TARGETNAME).$(EXT)" : "$(OUTDIR)" $(OBJS)
        $(LINK) @<< $(LINKFLAGS) /out:"$(OUTDIR)\$(TARGETNAME).$(EXT)" $(LINKOBJS) $(LINKLIBS)
<<

all : "$(OUTDIR)\$(TARGETNAME).$(EXT)"

clean::
        -@del $(LINKOBJS:/=\)
        -@rmdir /s /q $(OUTDIR)

rebuild : clean all
```

Notes:

1. Note the `TARGETTYPE` of `DLL` as a result of the `targetType` setting of "shlib" for the project.
2. The generated makefile contains all of the files we need to build the library.

Code Snip 4 - 9: The Makefile

4.2.5 Providing an Implementation

So where do we go from here? If you recall, the generated C++ peer contained skeleton methods with comments urging you to fill in the implementation. That is the next step, and the implementation of our simple example is shown below.

```
#include "SimpleDemoPeer.h"
#include "net/sourceforge/jnipp/JStringHelper.h"
#include <iostream.h>
```

```
   using namespace demo::chapters::shi::simple;
   using namespace net::sourceforge::jnipp;

   SimpleDemoPeer::SimpleDemoPeer()
   {
   }

1:

   // methods
   void SimpleDemoPeer::printMessage(JNIEnv* env, jobject obj, jstring p0)
   {
          // TODO: Fill in your implementation here
          cout << "The following message was printed from the C++ peer: " << JStringHelper( p0 ) <<
   endl;
   }
```

Notes:

1. Here is the generated method with implementation filled in.

Code Snip 4 - 10: The C++ Peer with Implementation

4.2.6 Building the Project

Now that the implementation has been provided, we can build the shared library using the generated makefile. As with the JNI++ Proxy Generator, the generated makefiles are dependent upon the `JAVA_HOME` and `JNIPP_HOME` environment variables being appropriately set and also `OS` if you are building under UNIX. See the first example of the previous chapter for details. The build output is shown below.

```
C:\development\jnipp\demo\chapters\shi\simple>nmake /f simple.mak CFG="Release" all

Microsoft (R) Program Maintenance Utility   Version 6.00.8168.0
Copyright (C) Microsoft Corp 1988-1998. All rights reserved.

        if not exist "Release\" mkdir "Release"
        cl.exe @C:\DOCUME~1\ptrewhel\LOCALS~1\Temp\nma00628. /nologo /GX /W3 /FD /D "WIN32" /D
"_WINDOWS" /D "_MBCS" /I .\ /I "c:\java\jdk1.3/include" /I "c:\java\jdk1.3/include/win32" /I
"c:\development\jnipp\dist/include" /c /D "_USRDLL" /MD /O2 /D "NDEBUG"
/Fo.\demo\chapters\shi\simple\SimpleDemoPeer.obj .\demo\chapters\shi\simple\SimpleDemoPeer.cpp
SimpleDemoPeer.cpp
        cl.exe @C:\DOCUME~1\ptrewhel\LOCALS~1\Temp\nmb00628. /nologo /GX /W3 /FD /D "WIN32" /D
"_WINDOWS" /D "_MBCS" /I .\ /I "c:\java\jdk1.3/include" /I "c:\java\jdk1.3/include/win32" /I
"c:\development\jnipp\dist/include" /c /D "_USRDLL" /MD /O2 /D "NDEBUG"
/Fo.\demo\chapters\shi\simple\SimpleDemoMapping.obj .\demo\chapters\shi\simple\SimpleDemoMapping.cpp
SimpleDemoMapping.cpp
        link.exe @C:\DOCUME~1\ptrewhel\LOCALS~1\Temp\nmc00628. /nologo /machine:IX86
/libpath:"c:\development\jnipp\dist/lib" /incremental:no /dll /out:"Release\SimpleDemo."dll""
.\demo\chapters\shi\simple\SimpleDemoPeer.obj .\demo\chapters\shi\simple\SimpleDemoMapping.obj
libJNIPPCore.lib
   Creating library Release\SimpleDemo.lib and object Release\SimpleDemo.exp

C:\development\jnipp\demo\chapters\shi\simple>
```

Output Sample 4 - 3: Building the Project with the Generated Makefile

4.2.7 Exercising the Generated Code

Similar to the way we tested in the previous chapter, all of the examples in this chapter will rely on user-supplied source code to exercise the generated code. The test code in the previous chapter was supplied in the various `Main.cpp` files. In this chapter, the driver code is written in Java and lives in the `main()` method of the `Main` class.

The test driver for this example simply creates an instance of a generated `SimpleDemoProxy` class and calls the `printMessage()` method defined in the input Java interface. The source is shown below.

```
   package demo.chapters.shi.simple;

   public class Main
   {
           public static void main(String[] args)
           {
1:
                   SimpleDemo sd = new SimpleDemoProxy();

                   sd.printMessage( "Hello JNI++ World!!" );
           }
   }
```

Notes:

1. Note that the `SimpleDemoProxy` instance is assigned to a `SimpleDemo` reference. Because it implements the `SimpleDemo` interface, it can be passed to any method that requires a `SimpleDemo` parameter. The implementation lives in the generated C++ peer class.

Code Snip 4 - 11: The Driver Code

4.2.8 Compiling the Java Sources

Both the generated Java proxy class and the user-supplied driver code just shown need to be compiled in order to run the test. Note in the output below that we are compiling the sources into a directory that lives in the `CLASSPATH`. The `-classpath` parameter is passed to the Java compiler so that the `SimpleDemo` interface can be resolved.

```
C:\development\jnipp\demo\chapters\shi\simple>javac -classpath %JNIPP_HOME%\build\classes -d
%JNIPP_HOME%\build\classes demo\chapters\shi\simple\SimpleDemoProxy.java

C:\development\jnipp\demo\chapters\shi\simple>javac -classpath %JNIPP_HOME%\build\classes -d
%JNIPP_HOME%\build\classes Main.java
```

Output Sample 4 - 4: Compiling the Java Proxy and Driver Code

4.2.9 Running the Finished Product

Running the user-supplied Java Main class from the command line yields the expected results shown below. Note that the `PATH` (or `LD_LIBRARY_PATH` under UNIX) must include not only the path to the core JNI++ run-time library (`%JNIPP_HOME%\lib`) but also the directory containing the native implementation library we just created (the `Release` directory shown below). The `CLASSPATH` must be set so that we can resolve the generated Java proxy class, the `Main` class and the input Java interface.

```
C:\development\jnipp\demo\chapters\shi\simple>set PATH=%PATH%;%JNIPP_HOME%\lib;Release

C:\development\jnipp\demo\chapters\shi\simple>java -cp %CLASSPATH%;%JNIPP_HOME%\build\classes
demo.chapters.shi.simple.Main
The following message was printed from the C++ peer: Hello JNI++ World!!

C:\development\jnipp\demo\chapters\shi\simple>
```

Output Sample 4 - 5: Running the Example

4.2.10 A Peek Under the Hood

So how does it all work? The generated C++ proxy class creates an instance of its Java peer on the other side of the JNI during a constructor call. Because we did not provide a constructor for our simple Java class, it was provided a default by the Java compiler. For each of the constructors of the input Java class, there is a corresponding constructor for the generated C++ proxy class. The C++ proxy class makes a `NewObject()` call in its constructors and passes in the appropriate parameters (in this case none). It then saves the `jobject` return value for use in subsequent method calls.

All of the rest is pretty straightforward, as illustrated in the following sequence diagram. Each of the methods of the input Java class results in a corresponding method in the generated C++ proxy class. When one of the generated proxy methods is called, the `methodID` is retrieved and utilized in the subsequent call to the corresponding method of the Java class instance.

The JNI++ Peer Generator generates a set of code that cooperates to map the Java proxy class and C++ peer. As shown in the sequence diagram below, the `static` block of the generated Java proxy class is invoked when the class is loaded. This causes the shared library (`SimpleDemo.dll` under Win32, `libSimpleDemo.so` under UNIX) to be loaded. After loading the shared library, the native `init()` method is called, resulting in the registration of the remaining `native` methods. Next, the `Main.main()` method creates an instance of the generated `SimpleDemoProxy` class. You might actually expect this to create an instance of the generated C++ peer class, but this is delayed until the first instance method is invoked, which is shown next. The `printMessage()` method call is mapped by the JNI to the generated mapping code as specified in the earlier `RegisterNatives()` call. The generated mapping code retrieves the `peerPtr` field value from the Java proxy instance, and if it is `NULL`, creates an instance of the peer class (through the generated factory) and sets the `peerPtr` value to the pointer of the new C++ peer instance. The method call is then delegated to the peer class instance where the implementation lives. Subsequent method calls will retrieve the `peerPtr` field value to locate the C++ peer class instance associated with the given Java proxy instance, and the lifetime of the peer instance is bound by the life of the associated Java proxy instance.

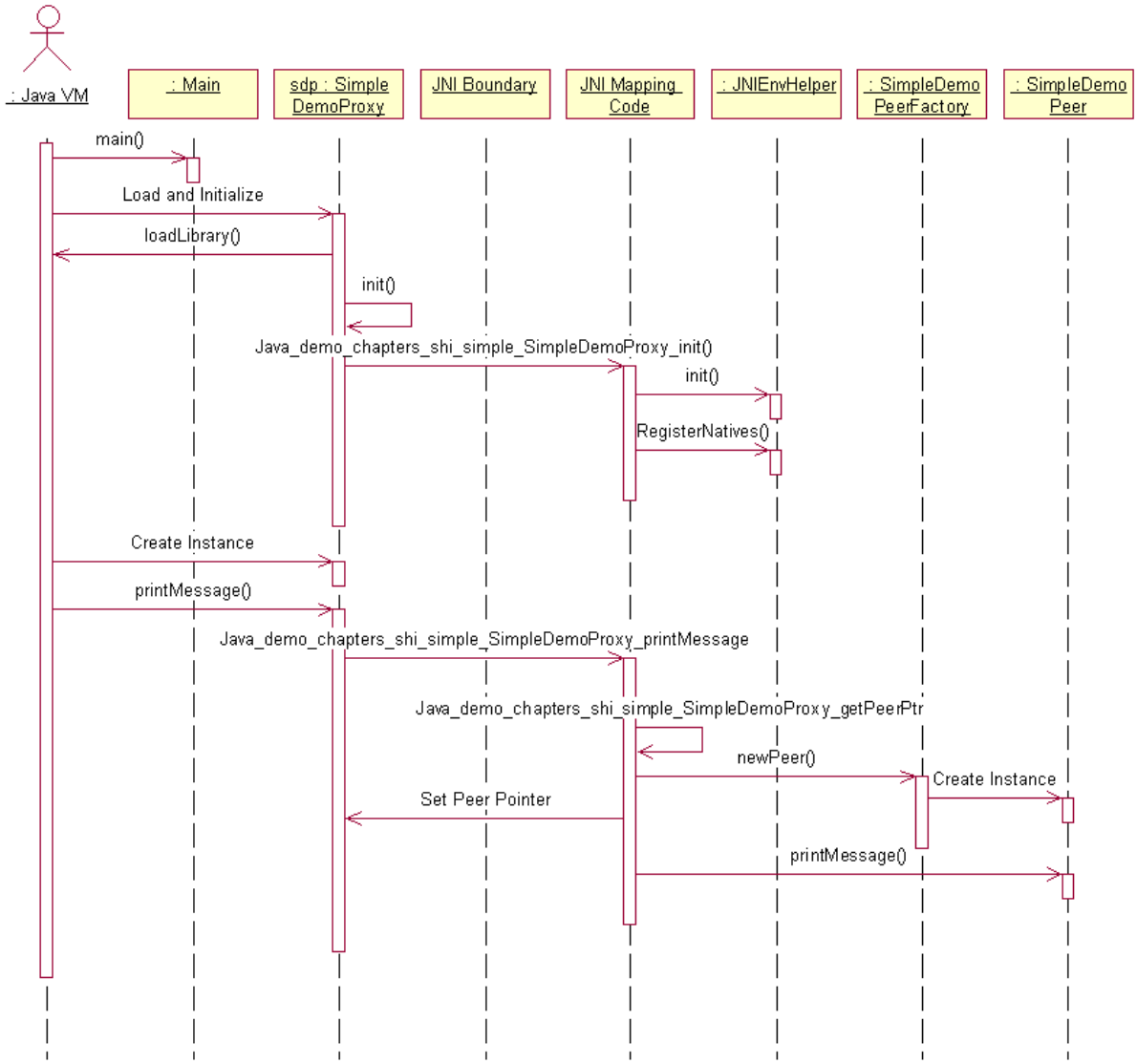

Figure 4 - 3: Sequence of Events for the Simple Demo

4.3 Derived Implementation

The previous example provided an in-line implementation for the generated C++ peer class and relied on non-destructive code generation, which sometimes requires a little hand-holding. As an alternative, the JNI++ Peer Generator can be instructed to generate a virtual base class with a default implementation of the interface methods. The implementation can then be provided in a derived implementation class.

This section will simply show the first example above but with a derived, rather than in-line, implementation. There is very little change required to provide a derived implementation, and these changes will be highlighted in this section.

4.3.1 The Input Java Interface

The same interface is used for this example, with the exception of the package and class names. The source is shown below.

```
package demo.chapters.shi.derived;

public interface DerivedDemo
{
        public void printMessage(String message);
}
```

Code Snip 4 - 12:

4.3.2 Project File Changes

The project file shown below is identical to that of the previous example with one exception. The `useInheritance` attribute is set to "true", and this signals the code generator to generator a virtual base class from which we will derive the implementation.

```
<?xml version="1.0"?>

<project name="derived" targetType="shlib" targetName="DerivedDemo">

1:      <peergen useInheritance="true">

                <input-classes>
                        <input-class name="demo.chapters.shi.derived.DerivedDemo"/>
                </input-classes>
        </peergen>
        <nmakefile name="derived.mak"/>
        <gnumakefile name="Makefile"/>
</project>
```

Notes:

1. We will provide a derived implementation. The default value is "false".

Code Snip 4 - 13: The Project File

4.3.3 The Generated Code

4.3.3.1 The Generated Peer Header

The header file for the generated C++ peer is shown below. With the `useInheritance` attribute set to "true", the JNI++ Peer generator generates all of the methods as `virtual`. The remaining code should look similar to the previous example.

```
#ifndef __demo_chapters_shi_derived_DerivedDemoPeer_H
#define __demo_chapters_shi_derived_DerivedDemoPeer_H


#include <jni.h>
#include <string>
namespace demo
{
```

```
        namespace chapters
        {
                namespace shi
                {
                        namespace derived
                        {
                                class DerivedDemoPeer
                                {
                                private:

                                public:
                                        DerivedDemoPeer();
1:
                                        // methods
                                        virtual void printMessage(JNIEnv* env, jobject obj, jstring
p0);

                                };
                        };
                };
        };
};

#endif
```

Notes:

1. Unlike the previous example, all of the generated methods are declared `virtual` so that the implementation can be deferred to a sub-class.

Code Snip 4 - 14: The Base Peer Header

4.3.3.2 The Generated Base Peer Implementation

Each of the methods of the input Java class results in a method in the generated C++ peer that contains a default implementation. The source for the generated implementation is shown below.

```
#include "DerivedDemoPeer.h"

using namespace demo::chapters::shi::derived;

DerivedDemoPeer::DerivedDemoPeer()
{
}
1:

// methods
void DerivedDemoPeer::printMessage(JNIEnv* env, jobject obj, jstring p0)
{
}
```

Notes:

1. Here is the default no-op implementation of the single input method.

Code Snip 4 - 15: The Base Peer Implementation

4.3.3.3 The Generated Peer Factory Header

If you recall from the previous example, the generated mapping code utilized a factory class to create instances of the C++ peer. We left the factory unmodified for that example, but for a derived implementation, the factory must be updated to return an instance of the C++ proxy *implementation* when the `newPeer()` method is called.

The modified `DerivedDemoPeerFactory` header file is shown below.

```
   #ifndef __demo_chapters_shi_derived_DerivedDemoPeerFactory_H
   #define __demo_chapters_shi_derived_DerivedDemoPeerFactory_H


   #include "DerivedDemoPeer.h"

1:
   #include "DerivedDemoPeerImpl.h"

   namespace demo
   {
           namespace chapters
           {
                   namespace shi
                   {
                           namespace derived
                           {
                                   class DerivedDemoPeerFactory
                                   {
                                   public:
                                           static inline DerivedDemoPeer* newPeer()
                                           {
                                                   /*
                                                    * TODO: Implement the factory method.  For example:
                                                    * return new DerivedDemoPeerImpl;
                                                    */
2:
                                                   return new DerivedDemoPeerImpl;

                                           }
                                   };
                           };
                   };
           };
   };


   #endif
```

Notes:

1. We must include the header file for our derived implementation ...
2. ... and supply the code to create the instance.

Code Snip 4 - 16: The Modified Peer Factory Header

4.3.4 Providing the Implementation

When using a derived implementation, we must supply a class that derives from the generated C++ peer and provide the implementation of the methods in that class. The new header file is shown below.

```
   #ifndef __demo_chapters_shi_derived_DerivedDemoPeerImpl_H
   #define __demo_chapters_shi_derived_DerivedDemoPeerImpl_H

   #include "DerivedDemoPeer.h"

   namespace demo
   {
           namespace chapters
           {
                   namespace shi
                   {
                           namespace derived
                           {
1:
```

```
                                class DerivedDemoPeerImpl : public DerivedDemoPeer

                                {
                                public:
2:
                                        void printMessage(JNIEnv* env, jobject obj, jstring p0);

                                };
                        };
                };
        };
};

#endif
```

Notes:

1. We derive publically from the generated `DerivedDemoPeer` class ...
2. ... and also provide implementations for all of the methods declared in the peer.

Code Snip 4 - 17: The Implementation Header

The implementation of the `DerivedDemoPeerImpl` class declared above is shown below. For this example, we simply print the `java.lang.String` passed in with a pre-pended message. This method will ultimately be called for any call to the Java proxy class `printMessage()` method.

```
#include "DerivedDemoPeerImpl.h"
#include "net/sourceforge/jnipp/JStringHelper.h"
#include <iostream.h>

using namespace demo::chapters::shi::derived;
using namespace net::sourceforge::jnipp;

1:

// methods
void DerivedDemoPeerImpl::printMessage(JNIEnv* env, jobject obj, jstring p0)
{
        cout << "The following message was printed from the C++ peer: " << JStringHelper( p0 ) <<
endl;
}
```

Notes:

1. Here is our single method implementation.

Code Snip 4 - 18: The Derived Implementation

The generated makefile must now be modified because we need to include our derived implementation class in the build. The modified makefile is shown below with highlighted changes.

```
!IF "$(CFG)" != "Release" && "$(CFG)" != "Debug"
!MESSAGE You must specify a configuration by defining the macro CFG on the command line. For example:
!MESSAGE
!MESSAGE NMAKE /f derived.mak CFG="Debug"
!MESSAGE
!MESSAGE Possible choices for configuration are:
!MESSAGE
!MESSAGE "Release"
!MESSAGE "Debug"
!MESSAGE
!ERROR An invalid configuration is specified.
!ENDIF

!IF "$(OS)" == "Windows_NT"
NULL=
```

```
!ELSE
NULL=nul
!ENDIF

TARGETTYPE=DLL
SOURCES= .\demo\chapters\shi\derived\DerivedDemoPeer.cpp
.\demo\chapters\shi\derived\DerivedDemoMapping.cpp

1:

SOURCES=$(SOURCES) .\demo\chapters\shi\derived\DerivedDemoPeerImpl.cpp

OBJS=$(SOURCES:.cpp=.obj)

CPP=cl.exe
INC=/I .\ /I "$(JAVA_HOME)/include" /I "$(JAVA_HOME)/include/win32" /I "$(JNIPP_HOME)/include"
CPPFLAGS=/nologo /GX /W3 /FD /D "WIN32" /D "_WINDOWS" /D "_MBCS" $(INC) /c
LINK=link.exe
LINKFLAGS=/nologo /machine:IX86 /libpath:"$(JNIPP_HOME)/lib"

!IF "$(TARGETTYPE)" == "DLL"
CPPFLAGS=$(CPPFLAGS) /D "_USRDLL"
EXT="dll"
LINKFLAGS=$(LINKFLAGS) /incremental:no /dll
!ELSEIF "$(TARGETTYPE)" == "CONSOLEAPP"
EXT="exe"
LINKFLAGS=$(LINKFLAGS) /subsystem:console /incremental:no
!ENDIF

!IF "$(CFG)" == "Debug"

CPPFLAGS=$(CPPFLAGS) /MDd /Zi /Od /D "_DEBUG"
OUTDIR=Debug
LINKFLAGS=$(LINKFLAGS) /debug
LINKOBJS=$(OBJS:.obj=_d.obj)
LINKLIBS=libJNIPPCore_d.lib

TARGETNAME=DerivedDemo_d

"$(OUTDIR)":
        if not exist "$(OUTDIR)\$(NULL)" mkdir "$(OUTDIR)"

.cpp.obj:
        $(CPP) @<< $(CPPFLAGS) /Fo$(<:.cpp=_d.obj) $<
<<

!ELSE

CPPFLAGS=$(CPPFLAGS) /MD /O2 /D "NDEBUG"
OUTDIR=Release
LINKOBJS=$(OBJS)
LINKLIBS=libJNIPPCore.lib

TARGETNAME=DerivedDemo

"$(OUTDIR)":
        if not exist "$(OUTDIR)\$(NULL)" mkdir "$(OUTDIR)"

.cpp.obj:
        $(CPP) @<< $(CPPFLAGS) /Fo$(<:.cpp=.obj) $<
<<

!ENDIF

"$(OUTDIR)\$(TARGETNAME).$(EXT)" : "$(OUTDIR)" $(OBJS)
        $(LINK) @<< $(LINKFLAGS) /out:"$(OUTDIR)\$(TARGETNAME).$(EXT)" $(LINKOBJS) $(LINKLIBS)
<<

all : "$(OUTDIR)\$(TARGETNAME).$(EXT)"

clean::
        -@del $(LINKOBJS:/=\)
        -@rmdir /s /q $(OUTDIR)

rebuild : clean all
```

Notes:

1. The derived implementation must also be compiled and linked in.

Code Snip 4 - 19: Modifying the Generated Makefile

4.3.5 Building and Running

Armed with our modified makefile, we invoke the Ant build tool with the appropriate parameters to build the native implementation library. The build output has been omitted to preserve space, however the output of the test run is shown below.

```
C:\development\jnipp\demo\chapters\shi>java org.apache.tools.ant.Main -buildfile derived.xml run
Buildfile: derived.xml

run:
     [exec] The following message was printed from the C++ peer: Hello JNI++ World!!

BUILD SUCCESSFUL

Total time: 1 second

C:\development\jnipp\demo\chapters\shi>
```

Output Sample 4 - 6: The Final Result

Although the amount of effort required up front is considerably more than with an in-line implementation, it ultimately will pay off if you make frequent changes to the input interface. The non-destructive code generation is problematic if methods of the input Java interface are removed, modified or rearranged. Providing a derived implementation sidesteps all of these potential problems and saves time in the long run.

4.4 Rich Data Types

Remember the "rich types" behavior supported by the JNI++ Proxy Generator? The JNI++ Peer Generator also has the ability to generate "rich types" as parameter and return types. Ultimately it will invoke the JNI++ Proxy Generator to do this work after generating peer classes for the input Java interfaces. This feature represents the ultimate in convenience and melds the capabilities of both of the code generators into a single, powerful tool.

This example will explore this capability by defining an input Java interface with a single method, `getCollection()`, that returns a `java.util.Iterator`. The implementation will, of course, be in C++, and we will soon see how using the "rich types" feature results in code that is not only easy to work with, but "feels" very similar to the same code written in Java. The source for the interface is shown below.

4.4.1 The Input Java Interface

```
package demo.chapters.shi.rich;

import java.util.Iterator;

public interface RichDemo
{
        Iterator getCollection();
}
```

Code Snip 4 - 20: The Java Interface

4.4.2 The Project File

Using rich type support with either the Proxy Generator or the Peer Generator necessitates setting the `usePartialSpec` project attribute to "false" if your compiler does not support partial template specialization. Set this to "true" for MS VC++, and omit it for GNU g++. The project file (targeted for Win32) is shown below.

```
   <?xml version="1.0"?>
1:
   <project name="rich" targetType="shlib" targetName="RichDemo" usePartialSpec="false">

2:
           <peergen useInheritance="true" useRichTypes="true">

                   <input-classes>
                           <input-class name="demo.chapters.shi.rich.RichDemo"/>
                   </input-classes>
3:
                   <proxygen useRichTypes="true"/>

           </peergen>
           <nmakefile name="rich.mak"/>
           <gnumakefile name="Makefile"/>
   </project>
```

Notes:

1. Just as with the "rich types" demo in the previous chapter, the `usePartialSpec` attribute must be set to "false" if the target compiler does not support partial template specialization.
2. Unlike the JNI++ Proxy Generator, the default value of `useRichTypes` is "false" when invoking the Peer Generator. We set it to "true" here to turn on this feature.
3. Here is our nested `proxygen` element, required when `useRichTypes` is set to "true". The Proxy Generator will be invoked with `useRichTypes` set to "true".

Code Snip 4 - 21: The Project File

Note the nested `proxygen` element above. This element is required when `useRichTypes` is set to "true" and indicates the options to use when the JNI++ Proxy Generator is invoked. The JNI++ Proxy Generator will be invoked with a collection of input classes that includes all of the parameter and return types for all of the methods of the input Java interfaces (in our case, `demo.chapters.shi.rich.RichDemo`). The nested `proxygen` element specifies that the Proxy Generator should also be invoked with "rich type" support, so not only will a C++ proxy class be generated for the single `java.util.Collection` return type, but also for all of the field, parameter and return types for the `java.util.Collection` interface, *recursively*.

4.4.3 Invoking the Code Generator

Invoking the code generator results in numerous files and the output has been omitted to save space. The usual build process is followed -- the Ant build tool is invoked with the `rich.xml` as a "buildfile" parameter and "generate" as the target.

4.4.4 Providing an Implementation

We specified in the project file that we would provide a derived implementation, and the header is shown below.

```
#ifndef __demo_chapters_shi_rich_RichDemoPeerImpl_H
#define __demo_chapters_shi_rich_RichDemoPeerImpl_H

#include "RichDemoPeer.h"

namespace demo
{
        namespace chapters
        {
                namespace shi
                {
                        namespace rich
                        {
                                class RichDemoPeerImpl : public RichDemoPeer
                                {
                                public:

1:
                                        ::java::util::IteratorProxy getCollection(JNIEnv* env, jobject obj);

                                };
                        };
                };
        };
};

#endif
```

Notes:

1. Note that we are using a generated C++ Proxy class as the return type of our generated *peer* implementation. Recall that this method is `virtual` in the generated base class.

Code Snip 4 - 22: The Implementation Header

Our derived implementation is shown below. Note the ease of use that the generated C++ proxy classes provide us. The `java::util::ArrayListProxy`, `java::util::ListProxy` and `java::lang::ObjectProxy` classes (along with numerous others) have all been generated for our use -- all we need to do is `#include` the header files and use them. The resulting code is easy to read, understand and work with.

```
#include "RichDemoPeerImpl.h"

#include "java/util/ArrayListProxy.h"
#include "net/sourceforge/jnipp/JStringHelperArray.h"
#include "java/lang/ObjectProxy.h"
#include "java/util/ListProxy.h"

using namespace demo::chapters::shi::rich;
using namespace java::util;
using namespace net::sourceforge::jnipp;
using namespace java::lang;

IteratorProxy RichDemoPeerImpl::getCollection(JNIEnv* env, jobject obj)
{
        ArrayListProxy alp;
        alp.add( ObjectProxy( JStringHelper( "I" ) ) );
        alp.add( ObjectProxy( JStringHelper( "love" ) ) );
        alp.add( ObjectProxy( JStringHelper( "JNI++" ) ) );

1:
        ListProxy lp( alp );
        return lp.iterator();

}
```

Notes:

1. Once again, we are using the "inheritance alternative" pointed out in the last chapter. The `ArrayListProxy` class has no `iterator()` method, but to get at it we need only create an instance of the `ListProxy` class using the `ArrayListProxy` as parameter. This is acceptable because `ArrayList` *implements* the `List` interface.

Code Snip 4 - 23: The Derived Implementation

4.4.5 Building the Project

Using the generated makefile, we build the shared library with the single generated peer and the numerous generated proxy classes. The output has been omitted, once again, to save space. The "native" target is specified to build the shared library.

4.4.6 Running the Finished Product

Using the Ant build tool, we invoke the build with the "run" target to produce the expected output shown below.

```
C:\development\jnipp\demo\chapters\shi>java org.apache.tools.ant.Main -buildfile rich.xml run
Buildfile: rich.xml

run:
     [exec] Collection Contents:
     [exec] I
     [exec] love
     [exec] JNI++

BUILD SUCCESSFUL

Total time: 1 second

C:\development\jnipp\demo\chapters\shi>
```

Output Sample 4 - 7: Running the Example

4.5 Option Summary

A number of options have been demonstrated for use with the JNI++ Peer Generator, and these options, with default values are summarized in the table below for quick reference.

Attribute Name	Required?	Default	Description
destructive	No	false	Instructs the code generator to either generate code destructively or non-destructively, and can be either "true" or "false". This parameter is ignored if the `useInheritance` attribute is set to "true".
useInheritance	No	false	Directs the code generator to generate `virtual` methods with a default no-op implementation so that the implementation can be provided in a derived class as opposed to in-line. This field can be either "true" or "false".
useRichTypes	No	false	Causes the Peer Generator to invoke the Proxy Generator for each parameter and return type in the input Java interfaces.

Chapter 5

The JNI++ Helper Classes and Core Library

5.1 Overview

Some of the examples we've seen thus far utilize the JNI++ helper classes to ease the burden of accessing and manipulating the various raw JNI data types. Additionally, *all* code generated using the JNI++ toolset is dependent upon the core C++ runtime library for basic services. This chapter will cover the capabilities and use of the various helper classes and core services.

5.2 The Primitive Array Helper Classes

The primitive array helper classes provide a simplified interface for working with primitive arrays of *any* dimension. For each of the primitive array types, two template classes are provided -- one for working with arrays greater than one dimension and the other specialized for the single dimension array. Together they provide a greatly simplified interface for working with JNI primitive arrays. Additionally, because the array dimensions are specified as a template parameter (and thus become part of the type), the primitive array helper classes are crucial to providing the ability to generate unique signatures when using the "rich type" support of JNI++.

5.2.1 A Code Example

The example we will cover here demonstrates the use of the `JIntArrayHelper` template class in various forms. Working with the array helper types for the other JNI primitive arrays is identical except for the type of primitive data referenced. For this reason, the example will show the use of only a single array helper class.

For this example, we will generate a C++ peer class that implements the following interface.

5.2.1.1 The Input Java Interface

```
package demo.chapters.go.primArr;

public interface PrimArrDemo
{
        public int[][] get2DArray();
        public void show3DArray(int[][][] arr);
}
```

Code Snip 5 - 1: The Java Interface

5.2.1.2 The Project File

The project file specifies the use of "rich types" for the JNI++ Peer Generator but not for the nested `proxygen` element. This has the effect of generating the method signatures in the C++ peer class using "rich types" (recall that this includes generated proxy classes as well as JNI++ helper classes), but not generating the code recursively.

```
<?xml version="1.0"?>

<project name="primArr" targetType="shlib" targetName="PrimArrDemo" usePartialSpec="false">
        <peergen useInheritance="true" useRichTypes="true">
                <input-classes>
                        <input-class name="demo.chapters.go.primArr.PrimArrDemo"/>
                </input-classes>

1:
                <proxygen useRichTypes="false"/>

        </peergen>
        <nmakefile name="primArr.mak"/>
        <gnumakefile name="Makefile"/>
</project>
```

Notes:

1. Although we are setting up to use "rich types", we do not need to generate a plethora of classes. The nested `proxygen` element specifies no rich types.

Code Snip 5 - 2: The Project File

5.2.1.3 Generating the Code

It is assumed that you are familiar with the prerequisites for and the process of generating the code. If this is unfamiliar territory, please consult one of the previous two chapters. The commands and output are omitted for brevity.

5.2.1.4 The Generated C++ Peer

The generated C++ peer header file is shown below to illustrate the effect of specifying "rich types" in the project file. The generated methods utilize JNI++ primitive array helper classes for primitive array types. Note the array dimensions specified as template parameters.

```
#ifndef __demo_chapters_go_primArr_PrimArrDemoPeer_H
#define __demo_chapters_go_primArr_PrimArrDemoPeer_H


#include <jni.h>
#include <string>
#include "net/sourceforge/jnipp/JBooleanArrayHelper.h"
#include "net/sourceforge/jnipp/JByteArrayHelper.h"
#include "net/sourceforge/jnipp/JCharArrayHelper.h"
#include "net/sourceforge/jnipp/JDoubleArrayHelper.h"
#include "net/sourceforge/jnipp/JFloatArrayHelper.h"
#include "net/sourceforge/jnipp/JIntArrayHelper.h"
#include "net/sourceforge/jnipp/JLongArrayHelper.h"
#include "net/sourceforge/jnipp/JShortArrayHelper.h"
#include "net/sourceforge/jnipp/JStringHelper.h"
#include "net/sourceforge/jnipp/JStringHelperArray.h"
#include "net/sourceforge/jnipp/ProxyArray.h"

// includes for parameter and return type proxy classes (arrays first)

namespace demo
{
        namespace chapters
        {
                namespace go
                {
                        namespace primArr
                        {
                                class PrimArrDemoPeer
                                {
                                private:

                                public:
```

```
                                        PrimArrDemoPeer();
1:
                                        // methods
                                        virtual ::net::sourceforge::jnipp::JIntArrayHelper<2>
get2DArray(JNIEnv* env, jobject obj);
                                        virtual void show3DArray(JNIEnv* env, jobject obj,
::net::sourceforge::jnipp::JIntArrayHelper<3> p0);

                                };
                        };
                };
        };
};

#endif
```

Notes:

1. Array helper classes are generated for primitive array parameter and return types when `useRichTypes` is set to "true".

Code Snip 5 - 3: The Generated C++ Peer Header

5.2.1.5 Supplying the Implementation

The derived implementation is shown below. In both of the generated methods, we utilize the simplified interface presented by the `JIntArrayHelper` class to make easy work of a complicated task.

```
#include <iostream.h>
#include "PrimArrDemoImpl.h"
#include "net/sourceforge/jnipp/JIntArrayHelper.h"

using namespace demo::chapters::go::primArr;
using namespace net::sourceforge::jnipp;

::net::sourceforge::jnipp::JIntArrayHelper<2> PrimArrDemoImpl::get2DArray(JNIEnv* env, jobject obj)
{
1:
        JIntArrayHelper<2> retVal( 3 );

        for ( int i = 0; i < 3; ++i )
        {
2:
                JIntArrayHelper<1> arr( 4 );

                for ( int j = 0; j < 4; ++j )
                        arr.setElementAt( j, i*j );
                retVal.setElementAt( i, arr );
        }

        return retVal;
}

void PrimArrDemoImpl::show3DArray(JNIEnv* env, jobject obj,
::net::sourceforge::jnipp::JIntArrayHelper<3> p0)
{
        cout << "Contents of Java 3D array:" << endl;
        for ( int i = 0; i < p0.getLength(); ++i )
        {
3:
                JIntArrayHelper<2> arr2D = p0.getElementAt( i );

                for ( int j = 0; j < arr2D.getLength(); ++j )
                {
                        JIntArrayHelper<1> arr1D = arr2D.getElementAt( j );
```

```
                        for ( int k = 0; k < arr1D.getLength(); ++k )
                                cout << "\t" << arr1D.getElementAt( k );
                        cout << endl;
                }
                cout << endl;
        }
}
```

Notes:

1. Our return array is two-dimensional with three elements.
2. Each element of the return array is a one-dimensional integer array with four elements.
3. The `getElementAt()` method of all but single-dimension array helpers returns an array helper of one less dimension -- an array of arrays.

Code Snip 5 - 4: The Derived Implementation

5.2.1.6 Exercising the Implementation

The code to demonstrate the helper classes at work is shown below in the `Main` Java class. After retrieving and printing the contents of the two-dimensional array constructed in the C++ peer implemenation, a three-dimension array is constructed and passed to the peer instance for display.

```
package demo.chapters.go.primArr;

public class Main
{
        public static void main(String[] args)
        {
                PrimArrDemo pad = new PrimArrDemoProxy();

                System.out.println( "Contents of C++ 2D array:" );
                int[][] arr2D = pad.get2DArray();
                for ( int i = 0; i < arr2D.length; ++i )
                {
                        for ( int j = 0; j < arr2D[i].length; ++j )
                                System.out.print( "\t" + arr2D[i][j] );
                        System.out.println();
                }

                int[][][] arr3D = new int[3][4][5];
                for ( int i = 0; i < 3; ++i )
                        for ( int j = 0; j < 4; ++j )
                                for ( int k = 0; k < 5; ++k )
                                        arr3D[i][j][k] = i+j+k;

                pad.show3DArray( arr3D );
        }
}
```

Code Snip 5 - 5: The Java Driver Code

5.2.1.7 Running the Code

Running the `Main` Java class produces the following output.

```
C:\development\jnipp\demo\chapters\go>java org.apache.tools.ant.Main -buildfile primArr.xml run
Buildfile: primArr.xml

run:
     [exec] Contents of C++ 2D array:
     [exec]     0       0       0       0
     [exec]     0       1       2       3
     [exec]     0       2       4       6
```

```
     [exec] Contents of Java 3D array:
     [exec]     0       1       2       3       4
     [exec]     1       2       3       4       5
     [exec]     2       3       4       5       6
     [exec]     3       4       5       6       7
     [exec]
     [exec]     1       2       3       4       5
     [exec]     2       3       4       5       6
     [exec]     3       4       5       6       7
     [exec]     4       5       6       7       8
     [exec]
     [exec]     2       3       4       5       6
     [exec]     3       4       5       6       7
     [exec]     4       5       6       7       8
     [exec]     5       6       7       8       9
     [exec]

BUILD SUCCESSFUL

Total time: 1 second

C:\development\jnipp\demo\chapters\go>
```

Output Sample 5 - 1: Output of the Test Run

5.3 The JStringHelper Class

5.4 The JStringHelperArray Class

5.5 The ProxyArray Class

5.6 The JVM Class

5.7 The JNIEnvHelper Class

Chapter 6

Threads and Exceptions