IN-61
376 557

Interfacing with Legacy Libraries using Remote Method Invocation

Scott M. Howard
Scott.Howard@msfc.nasa.gov
Home: (205)353-0110   Work: (205)461-4307   Fax: (205)461-4999

The assignment described was enough to make a neophyte Java developer bolt for the door: provide a remote method for use by an applet which invokes a native method that wraps a function in an existing legacy library. Mentally calculating the odds of making it to the parking lot, I discarded that option and indicated my willingness to assume responsibility for this task with an air of cautious confidence. The purpose of the remote method is to return an instance of a class object whose contents reflect the data structure returned by the legacy function. Little did I know what I was getting myself into...

Perhaps the most significant hurtle I had to overcome while working on this task was the lack of useful documentation to help direct me in my efforts. While embroiled in implementation, I spent an entire day poring through the RMI use group archive on Sun's web site searching for guidance to no avail. I would have spent the time wading through their JNI use group archive as well, but I couldn't seem to locate one. Subsequently, I made the decision to try to document my findings in order to assist others.

Before we start on the class design, let's look at what the existing legacy code does. The C function to be called, `Get_Legacy_Data`, consists of two steps: an ASCII file is read from the local disk and its contents are parsed into a `Legacy_Type` structure whose address is passed as an argument by the caller. Not much to it, really. The legacy code was compiled into a shared object library, `legacy.so`, using the IRIX 6.2 compiler and then loaded onto the Web server, a Silicon Graphics Indy station loaded with the IRIX 6.4 operating system.

As far as the class design is concerned, the first thing required is a class to act as a template for the data structure returned by the legacy function. This class, `JLegacy`, declares a series of `public` instance variables which correspond to the members of `Legacy_Type` and provides a parameterless constructor. This constructor is never called, not even by the native method which allocates the object for return to the remote method.

Next, the remote interface declaration for the remote object must be defined. A remote interface is a Java interface that extends the interface `java.rmi.Remote`, which is used exclusively to identify remote objects. The remote method defined by `JLegacyIF`, `getJLegacy`, returns a `JLegacy` instance and throws `java.rmi.RemoteException` which provides a mechanism to handle any failures.

Now that the remote interface has been defined, let's look at the design of the remote object, `JLegacyRO`. In order for `JLegacyRO` to implement `getJLegacy`, `JLegacyRO` must interface with the existing legacy code through a native method, `getN`. `getN` is declared in the `JLegacyRO` class but implemented in C, just like the legacy code. `getN` returns a `JLegacy` instance and is declared `static` since its implementation is the same for all instances of the `JLegacyRO` class.

`getN` is implemented in a native shared object library, `libJLEG.so`, that is loaded into the Java virtual machine at run time. `libJLEG.so` is loaded using a `static` initializer in the `JLegacyRO` class. Static initializers are executed once by the Java virtual machine when the class is first loaded. If `JLegacyRO` doesn't load the native library, an `UnsatisifiedLinkError` exception is thrown when `getN` is called. Failure to load `libJLEG.so` is established only by catching one of the exceptions thrown by `System.loadLibrary`. The library name provided is qualified by the Java virtual machine which prepends `lib` and appends the library extension `.so` for UNIX and `.dll` for Microsoft Windows.

`JLegacyRO` implements the method defined by `JLegacyIF` by calling `getN` and returning the `JLegacy` object returned by it. Nothing to it, right? Well... let's finish the `JLegacyRO` class before we call this one a wrap.

The JLegacyRO class exports itself by extending UnicastRemoteObject and calling the constructor of its superclass in its own constructor. In addition, UnicastRemoteObject redefines the equals, hashCode, and toString methods inherited from java.lang.Object for remote objects.

The first thing that the main method provided by the JLegacyRO class does is install RMISecurityManager to protect its resources from remote client stubs, during transactions. The RMISecurityManager provides the equivalent function of the applet security manager for remote object applications.

Next, the main method creates an instance of the JLegacyRO class and a remote object registry listening on a port number which is declared static final. The JLegacyRO class is the only application that will use this registry.

Finally, the main method binds the instance of the JLegacyRO class to a unique name in the remote object registry, making the object available to clients on other virtual machines. The name bound to the object is formed using the port number, the name of the remote object's host which is passed to the application as a command-line argument, and the String "JLegacyRO".

Before delving into the details of the native method, let's look at the last class: the client-side class which invokes the method on the remote object, JLegacyC. JLegacyC provides a parameterless constructor which is never intended to be called and a static method, get, which looks up the remote object in the registry created by JLegacyRO and retrieves a reference to JLegacyIF through which the remote method, getJLegacy, is invoked. The get method returns the JLegacy object returned by the remote method invocation.

These three classes and the interface were compiled into the same package. All classes, including the stub and skeleton created from the JLegacyRO class using the rmic compiler, were served from the Indy Web server. The environment settings are explained at the conclusion of this article.

The native method to be implemented is relatively straightforward. However, before we can discuss its details, we must establish its C prototype. The C header file which defines the prototype for the native method is generated using the javah tool with the -jni option on the compiled JLegacyRO class. Since the JLegacyRO class has been compiled into a package, the package name must be appended to the class name when javah is executed (e.g., javah -jni my.jlegacy.classes.JLegacyRO). The resulting header file will be prefixed with the package name (e.g., my_jlegacy_classes_JLegacyRO.h).

If you have read the Java Native Interface specification, you are already familiar with the method used by javah in composing native method names. If you haven't, I must warn you: it is **not** pretty. A native method name has the following signature: Java_<mangled fully-qualified class name>_<mangled method name>. I might add that the term "mangled" is actually used in the JNI specification. If the native method were an overloaded method, the name is further concatenated with __<mangled argument signature>. There's that mangled word, again. For further information on the Java virtual machine's type signatures, I recommend reading the JNI specification.

The JNI interface (or JNIEnv) pointer is always the first argument to a native method. The interface pointer points to a table of function pointers, each of which is a JNI function. In standard C, all JNI functions are called via this pointer (e.g., (env)->FindClass(env, "java/lang/String")). The JNIEnv structure is defined in C++ with inline functions which ultimately resolve to the same references as the standard C functions. Since the sole purpose of the JNIEnv pointer is to invoke the JNI functions and its syntax is well-defined, I wrapped all of the JNI functions used in this native method to promote greater readability and ease of maintainability.

The second argument to a native method varies depending on whether the method is declared `static` or not. If the method is nonstatic, the argument is of type `jobject` and is a pointer to the Java object which invoked the method. In this case, the method is declared `static` so the argument is of type `jclass` and is a pointer to the Java class which declared the method, the remote object class `JLegacyRO`.

Any arguments passed to the native method in its Java declaration follow the second argument in the function prototype. In this case, the method is declared with no arguments.

Remember that `getN` was declared as returning an instance of the `JLegacy` class? This is the `jobject` returned by the function in the C prototype. Briefly, the native method will retrieve the required data using the existing legacy function, instantiate the `jobject` to be returned, and populate it with the retrieved data.

First, the native method calls `Get_Legacy_Data` passing it a pointer to the `Legacy_Type` structure to be populated. Then , the fun begins...

Using the JNI `AllocObject` function, the native method allocates an object of the `JLegacy` class. Because the native method is declared `static` in the `JLegacyRO` class, the `jclass` argument passed to it is not the class for which an object is to be allocated, so the `jclass` must be established first using the JNI `FindClass` function. `FindClass` requires a fully-qualified class name (i.e., `my/jlegacy/classes/JLegacy`).

The `JLegacy` object is an example of a local reference, which means that its scope is for the lifetime of the native method and it is automatically freed by the Java virtual machine upon return. All objects passed into or returned from native methods are local references. Global references remain visible until they are freed.

Once the `JLegacy` object is returned, the native method must establish the field IDs for the `public` instance (nonstatic) variables within the Java object in order to access the variables, or fields. Fields are identified by the JNI using their symbolic names and type signatures.

Finally, the instance fields are set to the contents of the `Legacy_Type` structure returned by `Get_Legacy_Data` using the JNI `Set<type>Field` family of accessor routines, and the populated `JLegacy` object is returned to the interface implemented by `JLegacyRO`. Former C programmers should note that the `Set<type>Field` routines are provided only for the following primitives: `boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, and `double`; everything else is an `Object` of some sort.

In this case, a series of the members in the `Legacy_Type` structure returned by `Get_Legacy_Data` are `char` arrays or UTF-8 format in Java. The UTF-8 format encodes nonnull ASCII characters in the range 0x01 to 0x7F (hexadecimal) in a single byte. Characters above 0x7F are encoded using up to three bytes of storage.

The JNI `SetObjectField` function requires a native type for the value of the indicated field, so the `char` arrays must be converted to `java.lang.String` objects before setting their instance fields in the Java object. This translation may be performed using the JNI `NewStringUTF` function. Since there is a series of these instance fields to be set, the steps taken to do this are generalized into another function, `JL_SetStringField`.

In the event that an error condition arose during execution of the native method, the method would delete the local reference pointed to by the `JLegacy` object and return a null object to the interface implemented by JLegacyRO. Freeing the local reference is a habitual practice of mine when I write C code which is not actually required in Java. To me it's just good programming style.

Now let's make everything talk to each other...

First, let's discuss compiling getN into the native shared object library, libJLEG.so. In the makefile for libJLEG.so, legacy.so must be supplied as an argument to the link editor in order to resolve the symbol supplied by Get_Legacy_Data's object module for getN.

In addition, Java 3.1 (Sun 1.1.5) assumes the run-time linker to load n32 libraries. If you attempt to load an o32 native library from the JLegacyRO class, a fatal error will be returned by rld indicating that it can not successfully map the shared object name to the LD_LIBRARY_PATH despite the presence of the native library being located at a path specified by the environment variable.

To facilitate loading of an o32 library, two options are available. The first is to set the environment variable SGI_ABI to "-32" before starting JLegacyRO; the second is to pass the "-32" argument to the java interpreter when starting JLegacyRO.

On the Indy Web server, the LD_LIBRARY_PATH variable must include the path for libJLEG.so and legacy.so, as well as /usr/java/lib/sgi/green_threads. Apparently, the Java virtual machine for the Silicon Graphics platform uses the default Green threads package as its user threading model. The Green threads package maps all Java threads into a single native thread, prohibiting concurrent execution of multiple threads in a Java application.

In addition, the CLASSPATH variable on the Indy Web server must include the path which precedes the directory structure defined by the package into which the classes were compiled, in order for the Java interpreter to locate them.

Finally, the applet class was served from the Indy Web server by setting the CODEBASE attribute, accordingly, in the HTML file.

Well, I hope this answered more questions than it raised. I know I learned a lot while working on this task, I even learned some more while describing how I did it. I hope you did, too.

RMI-JNI Command-Line Summary

Although all of these classes were served from the Indy Web server, a summary of the command-line steps from a client/server perspective might be useful. In this context, client refers to the process (i.e., applet) invoking a method defined by a remote object and server refers to the remote object process. This summary illustrates the client and server classes running on different platforms to make clear on which platform each class belongs and on which platform each step takes place. The `rmic` compiler is used on the server to create the stub and skeleton classes; the stub class is copied to the client before run time. In addition, `javah` is used on the server to generate the header file which defines the C prototype for the native method declared by the remote object class; development of the source file which implements the C function is left to the user. The `make` of the native shared object library on the server is not illustrated, nor is browser startup on the client.

<u>Server</u>

```
>ls
JLegacy.java
JLegacyIF.java
JLegacyRO.java

>javac JLegacy.java

>javac JLegacyIF.java

>javac JLegacyRO.java

>ls
JLegacy.class
JLegacy.java
JLegacyIF.class
JLegacyIF.java
JLegacyRO.class
JLegacyRO.java

>rmic JLegacyRO

>ls
JLegacy.class
JLegacy.java
JLegacyIF.class
JLegacyIF.java
JLegacyRO.class
JLegacyRO.java
JLegacyRO_Skel.class
JLegacyRO_Stub.class
```

<u>Client</u>

```
>ls
JLegacy.java
JLegacyC.java
JLegacyIF.java

>javac JLegacy.java

>javac JLegacyC.java

>javac JLegacyIF.java

>ls
JLegacy.class
JLegacy.java
JLegacyC.class
JLegacyC.java
JLegacyIF.class
JLegacyIF.java
```

```
/**********************************************************************
 * LEGACY.h defines the legacy structures and the associated function prototypes
 */

.
.
.

typedef enum { LEGACY_P__A,
               LEGACY_P__B,
               LEGACY_P__C,
               LEGACY_P__D
} Legacy_P_Type

typedef enum { LEGACY_M__A,
               LEGACY_M__B,
               LEGACY_M__C,
               LEGACY_M__D,
               LEGACY_M__E,
               LEGACY_M__F,
               LEGACY_M__G,
               LEGACY_M__H
} Legacy_M_Type;

.
.
.

typedef struct {
        time_t              Timestamp;
        Legacy_P_Type       P_Type;
        unsigned char       Id;
        Legacy_M_Type       M_Type;
        char                String_A[5];
        char                String_B[5];
        char                String_C[5];
        char                String_D[5];
        char                String_E[5];
        char                String_F[5];
        char                String_G[5];
        char                String_H[5];
} Legacy_Type;

int Get_Legacy_Data ( Legacy_Type *legacy );

.
.
.
```

Listing 1 : LEGACY.h

RMI-JNI Command-Line Summary (continued)

Server

```
>javah -jni my.jlegacy.classes.JLegacyRO

>ls
JLegacy.class
JLegacy.java
JLegacyIF.class
JLegacyIF.java
JLegacyRO.class
JLegacyRO.java
JLegacyRO_Skel.class
JLegacyRO_Stub.class
my_jlegacy_classes_JLegacyRO.h

>java my.jlegacy.classes.JLegacyRO hostname &
JLegacyRO: creating registry
JLegacyRO: bound in registry
```

Client

```
>ls
JLegacy.class
JLegacy.java
JLegacyC.class
JLegacyC.java
JLegacyIF.class
JLegacyIF.java
JLegacyRO_Stub.class
```

```
/***********************************************************************
 * JLegacy.java provides a class for the legacy data. A populated instance of a JLegacy
 * object is returned by JLegacyC.get().
 */
package my.jlegacy.classes;

public class JLegacy implements java.io.Serializable
{
   public long     timestamp;
   public int      pType;
   public byte     id;
   public int      mType;
   public String   stringA;
   public String   stringB;
   public String   stringC;
   public String   stringD;
   public String   stringE;
   public String   stringF;
   public String   stringG;
   public String   stringH;

   public JLegacy() {}
}
```

Listing 2 : JLegacy.java

```
/***********************************************************************
 * JLegacyIF.java defines the method used to return a populated instance of a JLegacy
 * object from a remote object.
 */
package my.jlegacy.classes;

public interface JLegacyIF extends java.rmi.Remote
{
   public JLegacy getJLegacy() throws java.rmi.RemoteException;
}
```

Listing 3 : JLegacyIF.java

```
/*************************************************************************
 * JLegacyRO.java provides a remote object which returns a populated instance of a
 * JLegacy object.
 */
package my.jlegacy.classes;

import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;
import java.net.*;
import java.io.*;

public class JLegacyRO extends UnicastRemoteObject implements JLegacyIF
{
   // JLegacyRO listens on this port in the remote object registry
   public static final int RO_REGISTRY_PORT = 1099;

   // The host address of JLegacyRO
   private String host;

   // Native method declaration
   public static native JLegacy getN();

   // Static initializer
   static
   {
      // Load the native library which includes getN
      try {
         System.loadLibrary("JLEG");
      }
      catch (SecurityException    e) { e.printStackTrace(); }
      catch (UnsatisfiedLinkError e) { e.printStackTrace(); }
   }

   public JLegacyRO() throws RemoteException { super();}

   public JLegacy getJLegacy()throws RemoteException
   {
      JLegacy jleg = null;

      jleg = JLegacyRO.getN();

      return jleg;
   }
 // Application
 public static void main (String args[])
 {
   JLegacyRO remote = null;

   System.setSecurityManager(new RMISecurityManager());

   try
   {
     remote = new JLegacyRO();
   }
   catch (RemoteException e) { e.printStackTrace(); }

   if (remote != null)
   {
     if (args.length == 1)
     {
       // Get host address of remote object
       remote.host = args[0];

       // Start registry and register remote object
       try
       {
         System.out.println( "JLegacyRO: creating registry");

         /* Create registry listening on RO_REGISTRY_PORT. We can do this since this application
          * is the only one that's going to use this registry.
          */
         LocateRegistry.createRegistry(RO_REGISTRY_PORT);

         Naming.bind( "rmi://" + remote.host + ":" + RO_REGISTRY_PORT + "/JLegacyRO",
                      remote);

         System.out.println("JLegacyRO: bound in registry");
       }
       catch (Exception e) { e.printStackTrace(); }
     }
     else
     {
       System.out.println("usage: JLegacyRO host_address");
     } // if (args.length == 1)
   } // if ( remote != null )
 } // main
}
```

Listing 4 : JLegacyRO.java

```
/*************************************************************************
 * JLegacyC.java provides a static method which returns a populated instance of a
 * JLegacy object.
 */
package my.jlegacy.classes;

import java.applet.*;
import java.rmi.*;

public class JLegacyC implements java.io.Serializable
{
   public JLegacyC() {}

   public static JLegacy get(Applet parent)
   {
      JLegacy   jleg  = null;
      JLegacyIF ifc = null;

      try
      {
         ifc = (JLegacyIF)
               Naming.lookup(
                  "rmi://"+parent.getCodeBase().getHost()
                  +"/JLegacyRO");
         if (ifc != null) jleg  = ifc.getJLegacy();
      }
      catch(Exception e) { e.printStackTrace(); }

      return jleg;
   }
}
```

Listing 5 : JLegacyC.java

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

#ifndef _Included_my_jlegacy_classes_JLegacyRO
#define _Included_my_jlegacy_classes_JLegacyRO
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     my_jlegacy_classes_JLegacyRO
 * Method:    getN
 * Signature: ()Lmy/jlegacy/classes/JLegacy;
 */
JNIEXPORT jobject JNICALL Java_my_jlegacy_classes_JLegacyRO_getN
  (JNIEnv *, jclass);

#ifdef __cplusplus
}
#endif
#endif
```

Listing 6 : my_jlegacy_classes_JLegacyRO.h

```
/*********************************************************************
 * Java_my_jlegacy_classes_JLegacyRO_getN.c contains the native
 * function which retrieves the legacy data using the legacy library routine, instantiates
 * a JLegacy object, and populates the object with the legacy data.
 */

#include <unistd.h>
#include <jni.h>

#include "my_jlegacy_classes_JLegacyRO.h"
#include <LEGACY.h>

/* The Java Native Interface functions used by this native method were wrapped
 * to promote greater readability and ease of maintainability
 */
#define JNI_ALLOCOBJECT(class)        (*env)->AllocObject(env, (class))

#define JNI_DELETELOCALREF(ref)  (*env)->DeleteLocalRef(env, (ref))

#define JNI_FINDCLASS(name)          (*env)->FindClass(env, (name))

#define JNI_GETFIELDID(name, sig)   (*env)->GetFieldID(env,jlClass, (name),(sig))

#define JNI_NEWSTRINGUTF(bytes)   (*env)->NewStringUTF(env, (bytes))

#define JNI_SETBYTEFIELD(id, val)   (*env)->SetByteField(env,jObject, (id),(val))
#define JNI_SETINTFIELD(id, val)     (*env)->SetIntField(env,jObject, (id), (val))
#define JNI_SETLONGFIELD(id, val)   (*env)->SetLongField(env,jObject, (id),(val))
#define JNI_SETOBJFIELD(id, val)     (*env)->SetObjectField(env,jObject,(id),(val))

/* Prototypes of functions found only in this source code file */
int JL_SetStringField(JNIEnv   *env,
                jobject   jObject,
                jfieldID  jFieldID,
                const char *bytes);

/*
 * Class:    my_jlegacy_classes_JLegacyRO
 * Method:   getN
 * Signature: ()Lmy/jlegacy/classes/JLegacy;
 */

/*********************************************************************
 * Java_my_jlegacy_classes_JLegacyRO_getN retrieves the legacy data using the
 * legacy library routine, instantiates a JLegacy object, and populates the object with
 * the legacy data.  Failure is indicated by returning a null object.
 */
JNIEXPORT jobject JNICALL Java_my_jlegacy_classes_JLegacyRO_getN
                    (JNIEnv *env, jclass jClass)
{
  Legacy_Type       legacy;
  int               istat         = 0;
  jclass            jlClass       = NULL;
  jfieldID          timestamp_ID = NULL;
jfieldID          pType_ID   = NULL;
jfieldID          id_ID      = NULL;
jfieldID          mType_ID   = NULL;
jfieldID          stringA_ID = NULL;
jfieldID          stringB_ID = NULL;
jfieldID          stringC_ID = NULL;
jfieldID          stringD_ID = NULL;
jfieldID          stringE_ID = NULL;
jfieldID          stringF_ID = NULL;
jfieldID          stringG_ID = NULL;
jfieldID          stringH_ID = NULL;
jobject          jObject     = NULL;

istat = Get_Legacy_Data(&legacy);
if ( istat < 0 ) return NULL;

jlClass = JNI_FINDCLASS("my/jlegacy/classes/JLegacy");
if ( jlClass )
{
  jObject = JNI_ALLOCOBJECT(jlClass);
  if ( jObject )
  {
    /* Establish the field IDs */
    timestamp_ID = JNI_GETFIELDID("timestamp", "J");
    pType_ID     = JNI_GETFIELDID("pType", "I");
    id_ID        = JNI_GETFIELDID("id", "B");
    mType_ID     = JNI_GETFIELDID("mType", "I");
    stringA_ID   = JNI_GETFIELDID("stringA", "Ljava/lang/String;");
    stringB_ID   = JNI_GETFIELDID("stringB", "Ljava/lang/String;");
    stringC_ID   = JNI_GETFIELDID("stringC", "Ljava/lang/String;");
    stringD_ID   = JNI_GETFIELDID("stringD", "Ljava/lang/String;");
    stringE_ID   = JNI_GETFIELDID("stringE", "Ljava/lang/String;");
    stringF_ID   = JNI_GETFIELDID("stringF", "Ljava/lang/String;");
    stringG_ID   = JNI_GETFIELDID("stringG", "Ljava/lang/String;");
    stringH_ID   = JNI_GETFIELDID("stringH", "Ljava/lang/String;");

    /* Set the instance fields of the object to be returned */
    if (     timestamp_ID
       && pType_ID
       && id_ID
       && mType_ID
       && stringA_ID
       && stringB_ID
       && stringC_ID
       && stringD_ID
       && stringE_ID
       && stringF_ID
       && stringG_ID
       && stringH_ID )
    {
      JNI_SETLONGFIELD(timestamp_ID, legacy.Timestamp);
      JNI_SETINTFIELD(pType_ID, legacy.P_Type);
      JNI_SETBYTEFIELD(id_ID, legacy.Id);
      JNI_SETINTFIELD(mType_ID, legacy.M_Type);
```

```
            if (     JL_SetStringField(env, jObject, stringA_ID, legacy.String_A)
               && JL_SetStringField(env, jObject, stringB_ID, legacy.String_B)
               && JL_SetStringField(env, jObject, stringC_ID, legacy.String_C)
               && JL_SetStringField(env, jObject, stringD_ID, legacy.String_D)
               && JL_SetStringField(env, jObject, stringE_ID, legacy.String_E)
               && JL_SetStringField(env, jObject, stringF_ID, legacy.String_F)
               && JL_SetStringField(env, jObject, stringG_ID, legacy.String_G)
               && JL_SetStringField(env, jObject, stringH_ID, legacy.String_H))
            {
               /* Instance string fields of object have been set */
            }
            else
            {
               JNI_DELETELOCALREF(jObject);
               return NULL;
            }
         }
         else
         {
            JNI_DELETELOCALREF(jObject);
            return NULL;
         }
      } /* if ( jObject ) */
   } /* if ( jlClass ) */

   return jObject;
} /* End of Java_my_jlegacy_classes_JLegacyRO_getN */
```

```
/************************************************************************
 * JL_SetStringField sets the indicated field of the object to a java.lang.String object
 * constructed from the indicated char array.
 */
int JL_SetStringField(JNIEnv     *env,
                      jobject     jObject,
                      jfieldID    jFieldID,
                      const char *bytes)
{
   int      retval = 1;
   jstring jString;

   jString = JNI_NEWSTRINGUTF(bytes);
   if ( jString )
   {
      JNI_SETOBJFIELD(jFieldID, jString);
      JNI_DELETELOCALREF(jString);
   }
   else
      retval = 0;

   return retval;
} /* End of JL_SetStringField */
```

Listing 7 : Java_my_jlegacy_classes_JLegacyRO_getN.c