Putting a Java Interface on your C, C++, or Fortran Code

Abstract : The purpose of this report is to document some of the technical aspects of creating Java interfaces for codes written in languages other than Java. We outline a procedure where one separates the construction of the interface from the external codes with the introduction of an intermediate "wrapper" class. This intermediate class serves to isolate user interface details from the details of calling external routines. This intermediate class also facilitates the incorporation of external routines into Java based systems for distributed computing and/or visual programming.

Contents

Chris Anderson
Department of Mathematics
UCLA Los Angeles, CA 91555
7/15/97

These software components were developed in conjunction with the research supported by Air Force Office of Scientific Research Grant F49620-96-I-0327 and National Science Foundation/ARPA Grant NSF-DMS-961584

Introduction

While people are debating whether or not Java is good for computationally intensive tasks, the fact is that C, C++ and Fortran are the primary languages for those who do scientific/technical computing. It also seems unlikely that this situation will change in the near future. Unfortunately, C, C++ and Fortran do not contain (as Java does) standardized and platform independent constructs for creating user interfaces, managing threads, networking, and a variety of other tasks associated with creating "applications". Thus, there is interest in creating applications in which the user interface, or other "application packaging", is written in Java, but the core computational component is written in C, C++, or Fortran (see Figure 1). The purpose of this document is to describe, principally by means of an extended example, the process of creating a Java interface for a program written in C, C++ or Fortran.

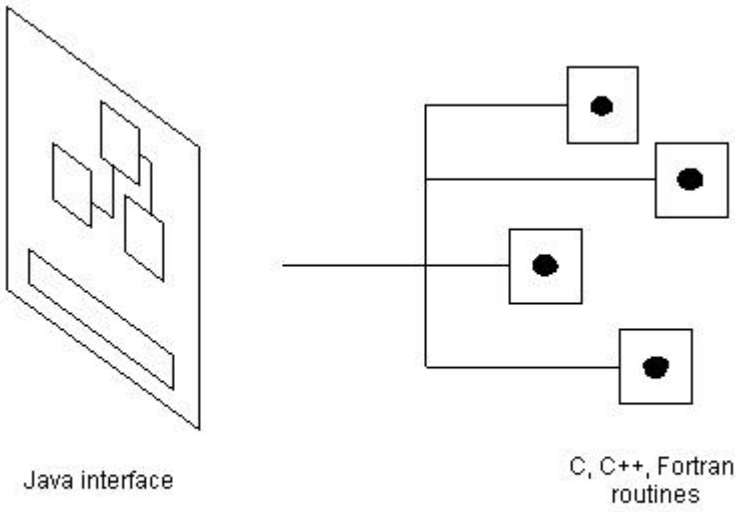


Figure 1

In order to create applications which have the form indicated in Figure 1, one needs to know how to write Java interfaces and how to call routines written in C, C++ and Fortran from Java. The process of writing a Java interface is well described in a variety of books [1][2][3] [4] and we will assume that the reader is capable of writing a modest Java interface which accepts input and displays output to a user. The task of calling routines written in C, C++ and Fortran from Java comes under the heading of implementing and using "native" methods. Here too, other documents [1][7] describe the process of interfacing Java to other languages. While, for completeness, we will outline the steps required to create and implement Java classes with native methods, we assume that the reader has implemented a Java class that has at least one native method procedure in it (e.g. the "Hello World" example of [7]).

In one aspect, this report is the presentation of an extended example demonstrating how this knowledge of writing Java interfaces and implementing native methods can be combined to create a Java/"other language" application. In addition to providing samples of the mechanisms for data exchange, the example also reveals the choices we made (and choices you will have to make) concerning the dividing line between the Java interface and routines written in C, C++ or Fortran. Our example concerns the creation of a Java interface for a program which solves the heat equation in a two dimensional rectangular region. Examples in C++ and Fortran are given (as is readily seen the C++ example is very close to what might be composed in C).

In the first section we outline the process that we follow for creating applications of the type described by Figure 1. In the second section we present the example which will form the basis of our discussion, and in the third and fourth sections we detail the construction

of the Java classes which form the primary components of the application.

The process of creating a Java interface to C, C++ and Fortran routines

The process that we use for creating Java interfaces consists of the three steps indicated in figure 2.

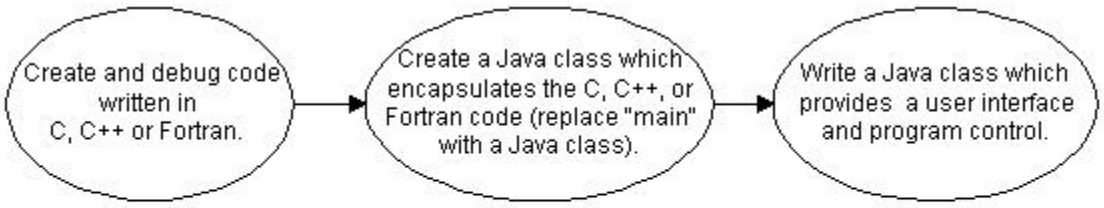


Figure 2 : Steps in the process of creating a Java interface.

A noticeable feature of the process is that we utilize three steps, rather than two. One may wonder about the need for the intermediate step; that of writing an intermediate class that ``wraps" the C, C++ or Fortran code. Originally we didn't have three steps, but adopted this practice for several reasons:

- It facilitated having the external code run as a separate thread. If one is running a computationally intensive task, then this allows the task to be executed without ``freezing" the interface.
- By using this intermediate class we have isolated that component of the application which contains inter-language calls. Since the inter-language calling procedure for Java is evolving, this allows us to accommodate any changes in the inter-language procedures more easily. Additionally, by not embedding this code within a user interface, we also allow the user interface to change independently (this is important because the Java user interface classes are evolving as well).
- Lastly, and no less importantly, this class provides an encapsulation of the external routines which facilitates their incorporation in a visual programming system or a software infrastructure which supports distributed computing.

The example program written in "another" language.

The starting point for the process of writing a Java interface is to have a program or a selected set of code components that one wishes to write interfaces for. Rather than

discuss the process of writing interfaces in an abstract way, we discuss the process of writing interfaces for a specific example. The example program is one that computes the evolution of the temperature of a rectangular plate. The main driver routine is given below (as well as in the file tempCalc.cpp); the include file for the functions which the main routine calls are given in tempCalcRoutines.h and the source for these routines is given in tempCalcRoutines.cpp .

In the first part of the main routine, the problem and run parameters are set, memory is allocated and the temperature distribution is initialized. A time stepping loop is then executed. In this loop, the temperature of the plate is evolved in time increments of size dt by calling the routine evolveTemperature(...) and at some predetermined number of time steps the temperature distribution is output. (In this case written to a file tempOut.dat).

Even though the temperature values are associated with a two-dimensional set of nodes covering the plate, we allocate and pass one-dimensional arrays of values. This was done because the standard method for exchanging data with other languages is through one-dimensional arrays; Java is no exception. Using one-dimensional sets of data values does not preclude using a two-dimensional array structure to access the data. The routines create2dArrayStructure(...) and destroy2dArrayStructure(...) in tempCalcRoutines.cpp demonstrate how one can create a two-dimensional array structure which access the data allocated as a one dimensional array.

```
#include <iostream.h>
#include <fstream.h>
#include "tempCalcRoutines.h"
void main()
{
//
// Set Problem parameters
//
   double diffusivity = 0.1;
   double a = 0.0; double b = 1.0;
   double c = 0.0; double d = 1.0;
//
// Set Runtime parameters
//
   long   m            = 10;
   long   n            = 20;
   long   nSteps       = 100;
   long   nOut         = 10;
   double dt           = 0.01;
//
// Allocate space for solution and work arrays
//
   double* Tarray     = new double[m*n];
   double* workArray  = new double[m*n];
//
// Open output file
//
```

```
   ofstream Fout("tempOut.dat");

   initializeTemperature(Tarray,m,n,a,b,c,d);
   double time        = 0.0;

   int i;  int j;

   for(i = 1; i <= nSteps; i++)
   {
   evolveTemperature(Tarray,m,n,a,b,c,d,dt,diffusivity,workArray);
   time = time + dt;

   if((i%nOut)== 0)
   {
      cout << " Step " << i << endl;      // print out step to screen

      Fout << m <<  "  "  << n <<  endl;  // output to file tempOut.dat
      Fout << time << endl;
      for(j = 0; j < m*n; j++){Fout << Tarray[j] << endl;}
   }

   }
   delete [] Tarray;
   delete [] workArray;
}
```

This program is typical of many computationally intensive applications; data is allocated, parameters and values are initialized, and then a time stepping loop is executed. As the calculation proceeds data is output periodically.

The Fortran version of this program is given in tempCalc.f and the supporting routines are given in tempCalcRoutines.f. One may notice that the C++ program is nearly identical to the Fortran program and does not use any of the object oriented features of C++ (i.e. it does not utilize classes). This was done intentionally so that the code would serve as an example of codes which are likely to be used (and/or written) by the majority of those involved in scientific/technical computation.

The Java class that encapsulates the C, C++ or Fortran codes components.

The second step in the process of creating an interface is to create a Java wrapper class that encapsulates the C, C++ or Fortran code components. It is in this class that the connection between the external routines and the corresponding Java routines is made. This class is also responsible for "loading" the external routines.

Essentially, this class replaces the main() routine. In this regard the class allocates the required arrays, contains the parameters as data members and also contains the methods (declared native) which are invoked by the main() driver routine (the initializeTemperature

and evolveTemperature routines).

To facilitate the execution of the program as a separate thread, this class implements the Runnable interface (it implements a run() method). In this run() method, we have changed the output process to be one which displays a color contour plot of the data, rather than write the output to a file. The requisite Java classes are contained in the files ColorContourPlot.java, ColorContourCanvas.java and ContourResolutionDialog.java.

Lastly this routine also includes a main routine of it's own for testing purposes. The complete code is given in TempCalcJava.java.

Since this routine has native methods, one must create the dynamically linked library (DLL) or shared library that contains their implementations. As outlined in the discussions on implementing native methods [1][7], this is a multi-step process:

1. The class TempCalcJava.java is compiled. Even though the native methods are not implemented, you must compile the Java class containing the native methods before performing the next step.

2. The command javah is applied to TempCalcJava.class. This means executing `"javah -jni TempCalcJava".` The result of this command is the creation of the file TempCalcJava.h. Since we are using the native interface specification of Java 1.1, the javah command must be the one distributed with the JDK 1.1.

3. The functions contained in TempCalcJava.h are "implemented". In this regard our task consists of accessing the data contained within the Java arrays and passing it to the corresponding C++ (or Fortran routines). The implementation of these routines is given in TempCalcJava.cpp. (Note that one can select the Fortran implementation by defining __FORTRAN_BUILD__ in the compilation process.)

4. The routines in TempCalcJava.cpp along with those in tempCalcRoutines.cpp are compiled and a dynamically linked library (or a shared library) is created. The name of this library must coincide with the name of the file (without the .dll or .so extension) which occurs in System.load or System.loadLibrary command within the static initializer for the class. (For some notes on the compilation process see Native Method Compilation Notes.)

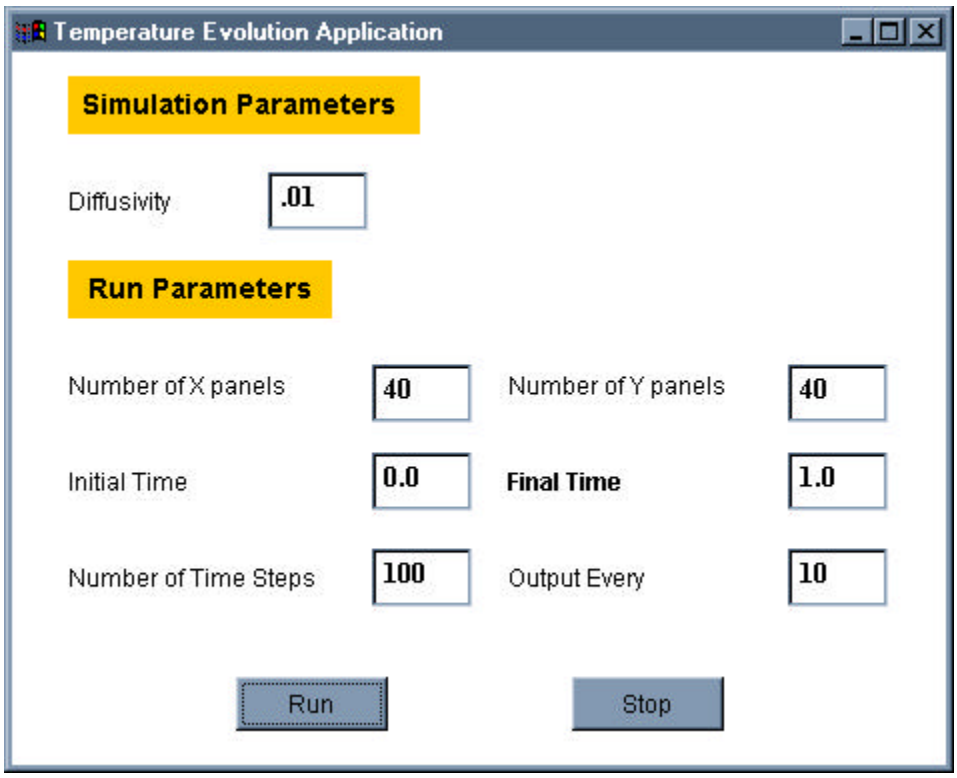
See "Native Method Implementation Process" for a diagram of these steps.

At this point, if the native method implementation process is successful, one should be able to run a "command line" version of the program by executing the main routine of the class i.e. just execute "java TempCalcJava". Problems which occur at this point are often caused by incorrect, or non-specification, of the path which is searched for the library containing the native method implementation. On PC/Windows platforms the PATH

variable must include the directory containing the native method implementation dll. On UNIX machines running solaris the LD_LIBRARY_PATH variable must include the directory containing the native method implementation shared library.

The Java Interface

The third step in writing the interface is to write the Java class that implements the interface. Minimally this means creating a Java application that possesses program control buttons and fields for data input. The interface is displayed below, and the associated Java code is contained within TemperatureApp.java.



This user interface was constructed using tools that generate Java 1.0.2. However, since our implementation of native methods is Java 1.1 based, after the initial construction, we compiled and worked with this code using the Java 1.1 compiler. In the Java 1.1 compilation step one must specify the flag "-deprication" and put up with all the warnings that are generated. Hopefully the interface construction tools will support Java 1.1 soon and these nuisances will disappear.

In looking over the Java code, one should take note that the computationally intensive part of the application is done as a separate thread [6]. Specifically, within the code which

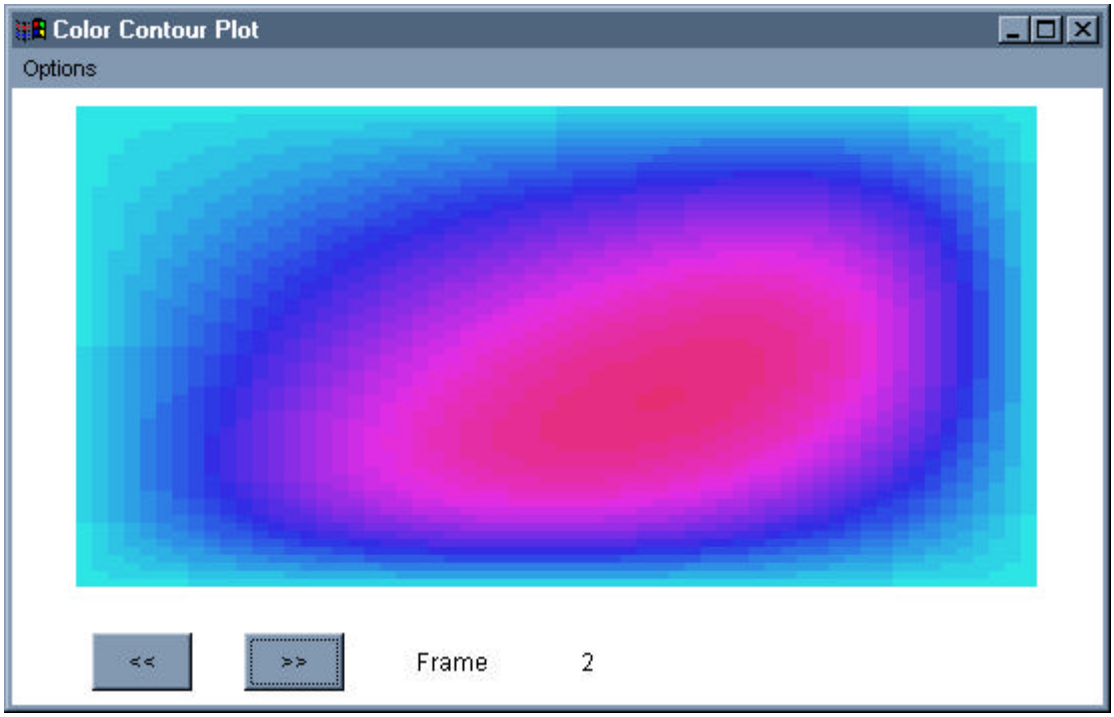
gets executed when the Run button is hit (the code fragment is given below) we create threads for the separate components of the application--- one thread for the calculation component and one thread for the color contour plot. The calculation component thread is given a lower priority, so that on machines whose implementation of the Java virtual machine doesn't time-slice among equal priority threads, the computationally intensive component will not cause the user interface to "freeze".

```
void RunButton_Clicked(Event event)
{
                        *
                        *
                        *
//
//  Set up and start the threads for the contour plot and the
//  calculation
//
    Thread current = Thread.currentThread(); // capture current thread

    Thread contourThread     = new Thread(temperatureRun.contourPlot);
    contourThread.start();

    TempRunThread = new Thread(temperatureRun);
    TempRunThread.setPriority(current.getPriority() -1);
    TempRunThread.start();
}
```

The color contour plot that results from the execution of the program is given below



References

1. Campione, M. and Walrath, K., "The Java Tutorial: Object-Oriented Programming for the Internet", Addison-Wesley, 1996.
2. Cornell, G. and Horstmann, C., "Core Java", SunSoft Press, 1996.
3. Daconta, C., "Java for C/C++ Programmers", Wiley Computer Publishing, 1996.
4. Flanagan, D. "Java in a Nutshell", O'Reilly and Associates, 1996.
5. Jackson, J. and McClellan, A., "Java by Example", SunSoft Press 1996.
6. Oaks, S. and Wong, H., "Java Threads", O'Reilly and Associates, 1997.
7. Campione, M. and Walrath, K, Integrating Native Code and Java Programs