Jace Developer's Guide − Cover

Table Of Contents

Chapter 1
Before You Begin

Getting Started

You can get the latest versions of this documentation at http://jace.reyelts.com/jace. An html version is available at http://jace.reyelts.com/jace/docs/guide/guide0.html and a pdf version is available at http://jace.reyelts.com/jace/docs/guide/guide.pdf.

If you're interested in learning how to develop with Jace, you've come to the right place, but first things first. If you've never used JNI before, you're going to be absolutely lost. No class library or set of tools can be an adequate substitute for the knowledge that you'll gain by reading through the freely available Java Native Interface Programmer's Guide and Java Native Interface Specification. Your time will have been well spent.

The Jace License

Jace is made available under the BSD license, which roughly means that you're free to do whatever you want to with it. Here's the fine print:

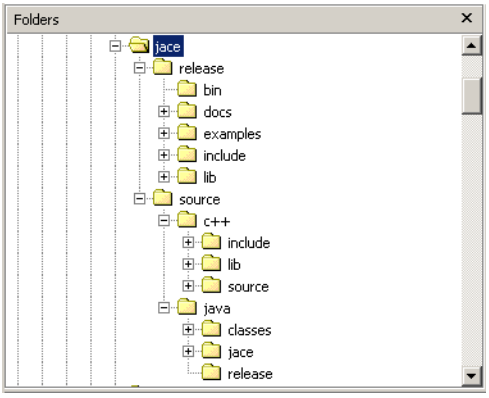
Compatibility

You've probably already got a development platform in mind, or perhaps two or three. Jace is excellent for cross−platform development, because it's built using only the standard C++ library and a JDK 1.2 or later compiler. Here's a matrix of all of the known Jace compatible compilers, operating systems, and architectures. If your favorite compiler/os/arch isn't listed, download the source and give the build a try. You're likely to experience a pleasant surprise.

	Windows i386	Solaris Sparc	Linux i386	MacOSX
Visual C++ 6.0	X			
Visual C++ .NET (7.0 and 7.1)	X			
Forte C++ 6.0		X		
GCC 3.x	X	X	X	X

Installation

If you haven't already, you may download Jace from http://jace.reyelts.com/jace or http://sf.net/projects/jace. You'll get a jace[XXX].jar file (where XXX is the version of Jace being downloaded) which you can then simply extract using your favorite zip tool (for example, Winzip) or the JDK's jar utility. This is what you should see:



The release folder contains all of the binaries and tools you need to use Jace. It consists of the following folders:

- bin − The proxygen, batchgen, autoproxy, and peergen code−generation tools.
- docs − The documentation necessary to use Jace − including what you're staring at right now.
- examples − Several example programs used to demonstrate the features available in Jace.
- include − The header files that you need to #include when you develop with Jace.
- lib − Debug and release binaries of the Jace Runtime Library (JRL) for a few platforms (for example, Windows and Linux).

The source folder contains the source used to build the JRL and code−generation tools. This is where you can go to build your own version of the JRL if Jace didn't come with pre−built binaries for your platform. The source directory has the following folders:

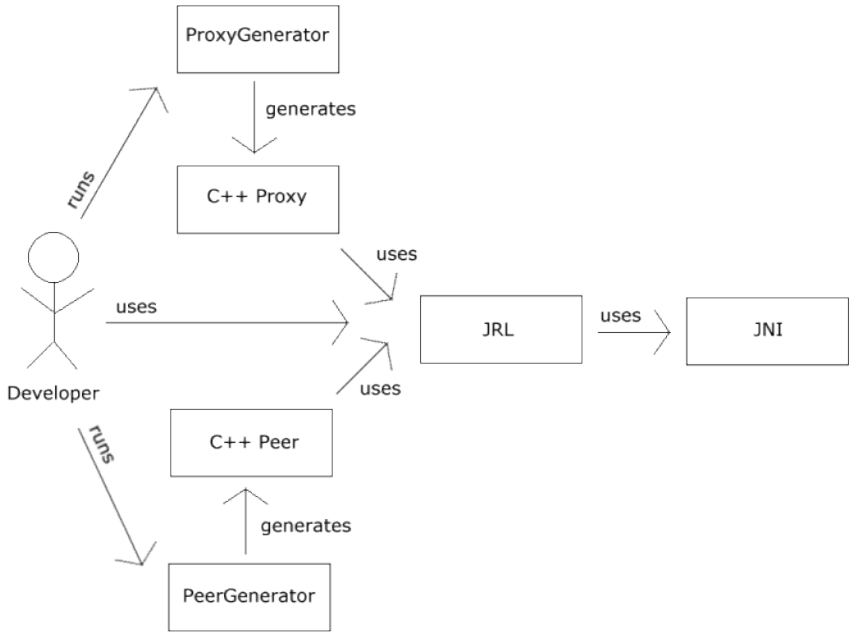
- c++/include − The C++ header files required to build the JRL.
- c++/lib − The makefiles and scripts used to build the JRL for several platforms.
- c++/source − The C++ source files required to build the JRL.
- java/jace − The Java source files required to build the entire set of code−generation tools.
- java/classes − An intermediary build directory for the Java classes.

Previous Next

Chapter 2
Introducing Jace

In 60 Seconds or Less

Jace is a C++ runtime library consisting of a single shared library named "jace" [1] and a set of Proxy and Peer code−generation tools: ProxyGenerator, BatchGenerator, AutoProxy, PeerEnhancer, and PeerGenerator. In addition to providing some useful utility functions for developers, The Jace Runtime Library (JRL) uses JNI to provide the basic services that the tool−generated Proxies and Peers require to run correctly. The generated Proxy classes allow developers to instantiate and manipulate live Java objects at runtime, just as if they were native C++ classes. The generated Peer classes provide an easy method for developers to implement native methods declared in their Java classes. The following diagram is a high−level overview of the relationships between a developer, the code−generating tools, and the JRL.



Previous Next

1) For example, "jace.lib" on Windows or "libjace.so" on Unix.

Chapter 3
Proxy Classes

Here a Proxy, There a Proxy, Everywhere a Proxy Proxy

Jace C++ Proxies are C++ classes that wrap existing Java types. If you examine the JNI type system, you'll see that there are a total of 24 different types.

nine primitive types:

- jboolean
- jbyte
- jchar
- jshort
- jint
- jlong
- jdouble
- jfloat
- void

fourteen reference types:

- jobject
- jclass
- jstring
- jthrowable
- jarray
- jobjectArray
- jbooleanArray
- jbyteArray
- jcharArray
- jshortArray
- jintArray
- jlongArray
- jfloatArray
- jdoubleArray

and the union type:

- jvalue

which can represent any of the primitive or reference types.

Jace models each JNI type with a corresponding C++ class as can be seen in Figure 1.

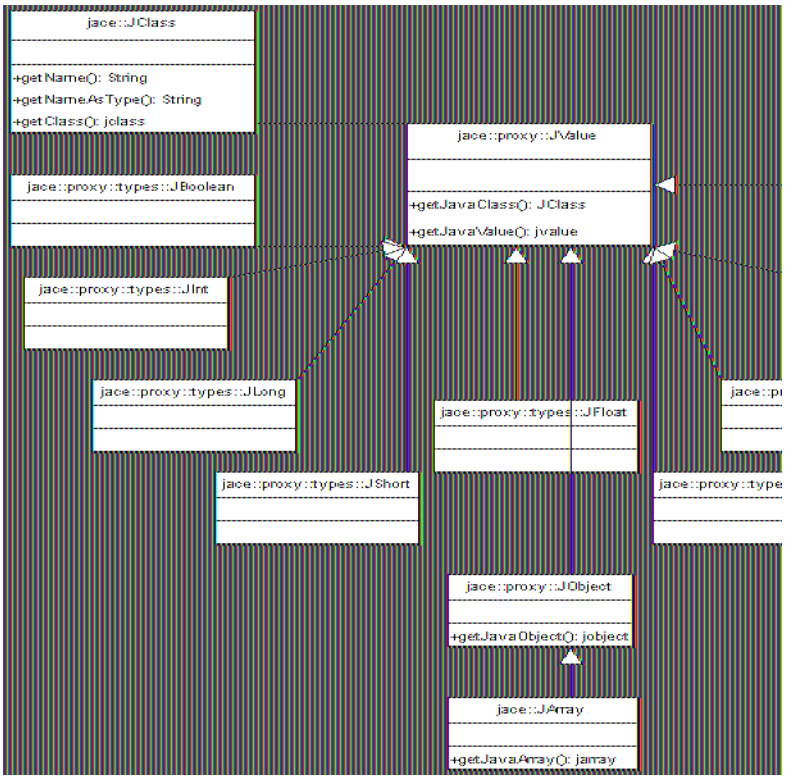


Figure 1

At the root of the class hierarchy is JValue which is the base class for every type in the system[1]. A JValue is mostly a simple wrapper around a JNI jvalue. Every JValue has a JClass which can be used to retrieve the JNI string which represents itself[2] and the JNI jclass that represents the Java type. JValues can be constructed with a JNI jvalue, and the JValue base class sets the rule that all of its subclasses must also be constructible with a jvalue.

Directly subclassing JValue, are the nine primitive Proxies. Primitive Proxies are simple wrappers around Java primitives, with convenient methods and overloaded operators to provide easy conversion with C++ primitives. The primitive Proxies can be instantiated from their matching JNI types or from their corresponding C++ types. For example, you can create a new JInt using a C++ int, or using a JNI jint:

```
jint jniInt = 32;
JInt fromJniInt( jniInt );
JInt fromCppInt( 32 );
```

Also subclassing JValue is JObject, the root of all Java reference types. JObject is a wrapper around the JNI jobject type, and as such, provides access to its JNI jobject, through the `getLocalObject()` method. Like JValues, JObjects can be constructed from jvalues, but they can also be constructed from jobjects. And also like JValue, JObject sets a rule for all of its base classes − only this time, the children must be constructible from a jobject as well as a jvalue. Differing from it's simpler ancestors and brethern, JObject has some added capabilities. First, when a JObject is constructed, it creates a new global reference to its jobject. You can access that global reference with a call to `getJavaObject()`. Second, you can test a JObject to see if it is null by calling, `isNull()`.

Subclassing JObject, is the template array class, JArray. The JArray class is a wrapper for all of the JNI array types: jarray, jobjectArray, jbooleanArray, jbyteArray, jcharArray, jshortArray, jintArray, jlongArray, jfloatArray, and jdoubleArray. The JArray class can take any Proxy class as a template parameter. You can construct a JArray either by passing it a JNI array or by specifying a size. For example,

```
void printLength( jintArray array ) {
  JArray<JInt> intArray( array );
  cout << intArray.length() << endl;
}
```

```
JArray<URL> createNewURLArrayOfLength( int length ) {
  JArray<URL> array( length );
  return array;
}
```

Like its Java counterpart, you can access elements of a JArray by using operator[](). For example,

```
void printArray( JArray<String> array ) {
  for ( int i = 0; i < array.length(); ++i ) {
    String str = array[ i ];
    cout << str << endl;
  }
}
```

Previous Next

1) (excepting JClass)

2) For example, the JClass for JInt returns "I", and the JClass for jace::proxy::java::lang::String returns "java/lang/String;"

Chapter 4
Peer Classes

Peer To Peer

Jace's C++ Peers are the yin to Jace's C++ Proxies' yang. Whereas the purpose of a C++ Proxy is to provide developers with easy access to a matching Java class from C++, the purpose of a C++ Peer is to provide developers with the capability to easily implement the native methods of a matching Java class.

As an example of Peers, consider the Java AWT Peer classes (Frame, Button, Choice, List, etc...). Each of these Peer classes has a corresponding native Peer class. When the Java Peer class is instantiated, it also instantiates the native Peer. When a native method is called on the Java Peer, the method call is passed onto the native Peer. When the Java Peer is destroyed (through a call to dispose() or a similar deallocation method), the native Peer is also destroyed.

Rather then delve deeply into the well known design pattern of Peer classes, I highly suggest that if you're not yet familiar with them, you read Section 9 of The Java Native Interface. This JDC tech tip is also a good tutorial on Peer classes.

With that background out of the way, I can now explain how Jace makes it easy to implement Peer classes. You just follow this simple recipe:

1. Write your Java Peer class as you normally would any other class. Make sure the methods that you want to implement in your native Peer are declared as native in your Java Peer.

2. Use the tool, PeerEnhancer, to enhance your Java Peer class to include all of the boilerplate Java Peer code you'd normally have to write by hand. The PeerEnhancer works by using BCEL to enhance the bytecode of your Java Peer class file to include new code, methods, and data. Once your Java Peer class has been enhanced, it will do several new things:

    - Load your native shared library upon class initialization of the Java Peer.
    - Create and store a matching native Peer when the Java Peer is constructed.
    - Destroy the matching native Peer when the Java Peer is garbage collected or when its deallocation method is called.

3. Use the tool, PeerGenerator to generate the boilerplate native Peer code for you. The PeerGenerator will generate several files:

    - A C++ header file which contains the declaration of your native C++ Peer class. This class contains the declarations for all of the Java Peer's fields and methods.
    - A C++ source file which contains the Jace implementation of all of the fields and methods declared in the C++ header file − except for the native methods. You will implement the native methods with your own code, and you have easy access to all of the Java Peer's fields and methods.
    - A C++ source file which contains all of the code required to map between the JNI native method calls and calls to the C++ Peer class. This is how calls from the JVM get translated to calls on your native C++ Peer.

4. Implement the native methods declared in the C++ Peer header. It's trivial to implement these methods, because they are declared totally in terms of C++ Proxy classes. Both the arguments and the return value are C++ Proxy classes. You can even throw C++ proxy exceptions, which Jace will catch and rethrow to the JVM as real Java exceptions.

5. Use the tool, AutoProxy to generate the necessary C++ Proxy classes. It will automatically generate all C++ Proxy classes that are required to implement your native Peer.

6. You're all done. Compile your code and take it for a spin.

Previous Next

Chapter 5
Virtual Machine Loading

Dynamic Statics

Jace provides several options for virtual machine loading. You can create a virtual machine from your C++ code by calling jace::helper::createVm() found in JNIHelper.h. There, you can specify a VmLoader, and a list of generic or virtual machine specific options.

To statically load a virtual machine, you must

- statically link with jvm.lib
- use StaticVmLoader in your call to createVm().

To dynamically load a virtual machine, you must

- *not* statically link with jvm.lib
- globally #define JACE_WANT_DYNAMIC_LOAD. This prevents StaticVmLoader from trying to statically bind with jvm.lib.
- use an appropriate dynamic VmLoader for your platform − for example, Win32VmLoader or UnixVmLoader. You may also write your own dynamic VmLoader if you so choose.

Whether or not you statically or dynamically load your virtual machine, you provide options to it via the same mechanism. You specify the entire set of virtual machine options in an OptionList which you pass in to the call to createVm().

Previous Next

Chapter 6
Tools

When all you know how to use is a hammer...

Jace comes with several tools which have already been mentioned in passing. This chapter documents each tool and its options in greater detail.

ProxyGenerator

```
Usage: ProxyGenerator <class file> <header | source> [ options ]
Where options can be:
  −protected : Generate protected fields and members
  −package : Generate package fields and methods.
  −private : Generate private fields and methods.
```

The ProxyGenerator generates the C++ Proxy class for a single Java class. You can specify whether it generates the header or the source file to standard output. You can also specify at which access level the ProxyGenerator generates member fields and methods. By default, the ProxyGenerator only generates public members. Normally, developers should prefer using the AutoProxy in preference to the ProxyGenerator, as the AutoProxy will walk the dependency tree to generate all dependee classes.

AutoProxy

```
Usage: AutoProxy
  <c++ header directory>
  <c++ source directory>
  <destination proxy header directory>
  <destination proxy source directory>
  <java classpath for proxies>
  [−mindep]
  [−deplist=<comma−separated list of classes>]
```

This tool scans c++ header files and source files for #includes that reference Jace C++ Proxies. It then generates the header and source files for the entire dependency tree for those C++ Proxies. For example, upon seeing a #include for jace/proxy/java/lang/RuntimeException.h, AutoProxy would generate the Proxy classes for RuntimeException, Throwable, Object, Serializable, String, etc... AutoProxy is also used in conjunction with other tools like BatchGenerator and PeerGenerator. You can use the recommended new option, "−mindep" to limit the number of proxy classes that AutoProxy generates. You can also optionally specify an additional list of classes for AutoProxy to process with "−deplist".

BatchGenerator

```
Usage: BatchGenerator <jar or zip file containing classes>
                      <destination directory for header files>
                      <destination directory for source files>
                      [ options ]
Where options can be:
  −protected : Generate protected fields and methods.
  −package : Generate package fields and methods.
  −private : Generate private fields and methods.
```

The BatchGenerator is used to generate all of the Proxy C++ classes inside of a jar file. This tool is useful when you are trying to create a C++ API for an existing Java API. You simply jar up all of your Java classes and then run BatchGenerator on the jar file. You will also want to run AutoProxy on the resulting C++ Proxy classes, so that all dependee C++ proxy classes are generated.

PeerEnhancer

```
Usage: PeerEnhancer
  <source path>
  <output path>
  <library>
  <deallocation method>
```

The PeerEnhancer is used to enhance the bytecode of Java Peer classes with automatic lifetime management code for native C++ Peers. The source path is the path to the class file to be enhanced. The output path is the path where the new class file will be written (It is recommended that these not be the same). The library is the name of the shared library which the newly enhanced Java Peer will try to load in its static initializer. The deallocation method is the name of the (already existing) resource deallocation method (for example, "close" or "dispose"), which will be enhanced to also deallocate the native C++ Peer.

BatchEnhancer

```
Usage: BatchEnhancer
  <library>
  <deallocation method>
  −sources <source files>
```

The BatchEnhancer runs the PeerEnhancer on multiple sources in a single run.

PeerGenerator

```
Usage: PeerGenerator <class file> <library>
<destination_header_directory> <destination_source_directory>
```

```
<user_defined_members = {true|false}>
```

The PeerGenerator generates the C++ header file and source code required to implement the native C++ Peer for a Java Peer. The class file is the path to the Java Peer's class file. The library is the name of the shared library which will contain this C++ source code. The destination_header_directory is the directory where the header containing the declaration of the C++ Peer class will be written. The destination_source_directory is the directory where the C++ source code for the JNI mappings and the C++ Peer implementation will be written. If you set user_defined_members to true, the C++ header file will #include an additional user header file, <peer_class_name>_user.h, where you can include any additional data or function members which you might require to implement the C++ Peer.

Previous Next

Chapter 7
Hello World

A simple example

The most tried and true method of explaining concepts is through demonstration, which is what the following chapters are all about. In this first section, I'm going to explain how example1 works. First, we'll examine, example1.cpp line by line. We begin with the #include's:

```
#include "jace/JNIHelper.h"

#include "jace/StaticVmLoader.h"
using jace::StaticVmLoader;

#include "jace/OptionList.h"
using jace::OptionList;

#include "jace/JArray.h"
using jace::JArray;

#include "jace/JNIException.h"
using jace::JNIException;

#include "jace/proxy/java/lang/String.h"
#include "jace/proxy/java/lang/System.h"
#include "jace/proxy/java/io/PrintWriter.h"
#include "jace/proxy/java/io/IOException.h"
#include "jace/proxy/java/io/PrintStream.h"

using namespace jace::proxy::java::lang;
using namespace jace::proxy::java::io;

#include <string>
using std::string;

#include <exception>
using std::exception;

#include <iostream>
using std::cout;
using std::endl;
```

All generated Jace Proxies are placed into the jace::proxy namespace under their Java package. For example, as can be seen above, the C++ Proxy for java.lang.String is located in the jace::proxy::java::lang namespace. Since this code will also be making use of the java.lang.System and java.io.PrintWriter Java classes, it also includes their C++ Proxies. Jace can also generate Proxies for Java exception classes. In this case, we're #including IOException, because we need to catch it if an exception is thrown. Following that, we #include JNIHelper.h which declares the utility functions that the JRL makes available for developers. If you examine, JNIHelper.h you will see that the utility functions are declared in the jace::helper namespace. To create a virtual machine, we need to specify a loader and virtual machine options, so we #include StaticVmLoader.h and OptionList.h Finally, we #include the string, exception, and iostream classes for use in this example.

```
int main() {

  try {
    StaticVmLoader loader( JNI_VERSION_1_2 );
    OptionList list;
```

```
    list.push_back( jace::CustomOption( "−Xcheck:jni" ) );
    list.push_back( jace::CustomOption( "−Xmx16M" ) );
    jace::helper::createVm( loader, list, false );
```

Here we define the standard C++ main() function. The very first step we take is to create a new Java Virtual Machine using the helper function, createVM(). For this example, we statically link to the JVM library, so we need to use the StaticVmLoader. Here we also demonstrate the use of CustomOptions to turn on extra JNI checking and a 16M heap limit. If we were to create the virtual machine on our own by not using createVm(), we would need to make sure to call jace::helper::setVmLoader() so that Jace is able to locate the virtual machine.

```
    for ( int i = 0; i <1000; ++i ) {
```

We run this code in a for loop to demonstrate that Jace is managing local references. Jace programs can run indefinitely, because it automatically deletes local references making objects available for garbage collection.

```
    String s1 = "Hello World";
    cout << s1 << endl;
```

The first line creates a new live java.lang.String object in the JVM. This line demonstrates that you can create a String from a C++ char*. The second line prints the String out to the console. The String class overloads operator<<() to write the result of toString() to the output stream. In fact, you can use operator<<() on any Jace Proxy class to write it out to an output stream.

```
    String s2 = std::string( "Hello World" );
    cout << s2 << endl;
```

These two lines of code do the exact same thing, only this time, the String is being constructed by a std::string instead of a char*. You can get a std::string from a String by calling String::operator std::string(). Take a look at String.h to see all of the ways that you can use String Proxies with C++ strings.

```
    String s3( "Hello World" );
    PrintStream out( System::out() );
    out.println( s3 );
```

Guess what this code does... You've got it − It also prints out "Hello World". This time, though, it uses Java's System.out static PrintStream member to print to standard out. With Jace, you can access class fields by calling a member function of the same name. In this case, we want to access System.out, so we call System::out().

```
    PrintWriter writer( System::out() );
    writer.println( "Hello World" );
    writer.flush();
```

I bet you would have never imagined there were so many different ways to do "Hello World". In this final example, we create a new PrintWriter, again using System.out. Also notice how the char*, "Hello World", is automatically converted to a java.lang.String − without any effort on our part.

```
     cout << i << endl;
    }

    return 0;
  }
  catch ( IOException& ioe ) {
```

```
    cout << ioe << endl;
    return −1;
  }
```

If the JVM throws an IOException during any of this code, the JRL will automatically catch it, clear the pending exception, locate a matching C++ Proxy class, and throw it. Jace Proxy exceptions are just like any other Jace Proxy class, except that they ultimately derive from std::exception in addition to Object.

```
  catch ( JNIException& jniException ) {
    cout << "An unexpected JNI error occured." << jniException.what() << endl;
    return −2;
  }
```

If any kind of JNI exception occurs during execution, the JRL detects it and throws a JNIException.

```
  catch ( std::exception& e ) {
    cout << "An unexpected C++ error occured." << e.what() << endl;
    return −3;
  }
```

Finally, every good C++ developer is going to make sure that no uncaught exception escapes his program.

Generating the Proxies

You'll notice that example1 makes use of many C++ Proxies. You could run the ProxyGenerator for each and every C++ proxy you need to generate. That can become very tedious, though. Especially considering that you'll need to also generate all of the dependee classes. (For example, IOException depends upon Exception, so you would also have to generate Exception). To avoid all of this hassle, you can use the AutoProxy. This wonderful little utility searches through your C++ header and source files looking for #includes of C++ Proxies. It then generates all of those Proxies and all of their dependent Proxies. I generated all of the Proxies for example1 (in the Proxies directory) by running

```
  autoproxy
    C:\data\projects\jace\release\examples\example1\include
    C:\data\projects\jace\release\examples\example1\source
    C:\data\projects\jace\release\examples\example1\proxies\include
    C:\data\projects\jace\release\examples\example1\proxies\source
    C:\java\jdk1.4\jre\lib\rt.jar
    −mindep
```

Well, actually, I just have an ANT script, build.xml which does that for me.

This tells the AutoProxy to recursively scan the example1\include and example1\source directories for proxy #includes, to generate the C++ headers and source files to proxies\include and proxies\source, and to use the JDK's rt.jar as the classpath to search for class definitions for the Proxies.

Building and running

In order to build example1, you'll need to adjust the Makefile settings so that the include and library directories point to your installation of the JDK. After that, you can run the build and generate the example1 program for your own platform. To run example1, you will need the JVM library to be in your path (for example, put jvm.dll in your PATH on Windows, or libjvm.so in your LD_LIBRARY_PATH on Linux or Solaris). If everything runs correctly, you'll get the output:

```
Hello World
Hello World
Hello World
Hello World
```

Generally speaking, whenever you build with the Jace library, you'll need to enable RTTI and multithreading for your compiler. For VC++ in particular, jaced.lib was built using the "Multithreaded Debug DLL" and jace.lib was built using the "Multithread DLL" options. You'll have to use those exact same options when you link with those libraries.

Previous Next

Chapter 8
Hello Peer

Moving onto Peers

This section covers a demonstration of the peer capabilities of Jace. The example, peer_example1, contains a single Java Peer class PeerExample.java which, as you can see, looks like any other Java class. It has one native method, getResources(), that we implement with a Jace C++ Peer. The first step in implementing the Peer is to compile and enhance PeerExample.class. If you examine build.xml you can see that after compiling PeerExample, we run the PeerEnhancer on the PeerExample.class and place the enhanced Java Peer class into the enhanced folder. Next, we run the PeerGenerator on the enhanced PeerExample.class. The PeerGenerator generates:

- The header file containing the declaration for the C++ Peer class − PeerExample.h (All peers are placed under the jace::peer namespace − Just like proxies are placed under the jace::proxy namespace).
- The mapping file containing the code that maps JNI function calls to the C++ Peer's methods − PeerExampleMappings.cpp and
- The source file containing the implementation for all of the members of the C++ Peer class − PeerExample.cpp. This file does not contain the native methods which must be implemented by the developer.

Next, we would actually implement the native methods. In this case, it's already been done in PeerExampleImpl.cpp. Next, we run the AutoProxy on the generated files and our own source code, so that all of the necessary C++ Proxy classes are created. Finally, we compile and build all of the source code.

Building and running

You can use ant to both build and run this example.

Chapter 9
A VM Loading Example

Loading done your way

This section demonstrates how you can use Jace to statically or dynamically load your virtual machine. The example, vm_load_example only has a single source file, main.cpp, which demonstrates how to load a virtual machine. As before, we'll examine the source code line by line:

```
#include "jace/JNIHelper.h"

#include "jace/OptionList.h"
using jace::OptionList;
using jace::Option;
using jace::Classpath;
using jace::Verbose;
using jace::CustomOption;

#include "jace/StaticVmLoader.h"
using jace::StaticVmLoader;

#ifdef _WIN32
  #include "jace/Win32VmLoader.h"
  using jace::Win32VmLoader;
#else
  #include "jace/UnixVmLoader.h"
  using ::jace::UnixVmLoader;
#endif

#include
using std::cout;
using std::endl;
```

It is JNIHelper.h that contains the function, createVm(), necessary to load and instantiate the virtual machine. When you call createVm(), you can specify the list of Options to be used in the creation of the virtual machine. All of the different Option types are defined in OptionList.h. When you call createVm(), you must also specify a VmLoader, which has the responsibilty of loading the virtual machine library and resolving functions that Jace requires to work with the virtual machine. StaticVmLoader is the default VmLoader, and works by statically binding to the JVM library. Win32VmLoader is able to search the registry for, and dynamically load virtual machines on the Windows platform. UnixVmLoader is able to dynamically load virtual machines on generic Unix platforms (those supporting dlopen()).

```
int main( int argc, char* argv[] ) {

  #ifdef JACE_WANT_DYNAMIC_LOAD

    if ( argc != 2 ) {
    cout << "Usage: vm_load_example " << endl;
      return −1;
    }

    string path = argv[ 1 ];
```

To turn on dynamic loading, you must globally #define JACE_WANT_DYNAMIC_LOAD. This keeps the StaticVmLoader from trying to statically bind to a JVM. Here, we check to see if

JACE_WANT_DYNAMIC_LOAD is defined. If it is, then we let the user specify the path to the JVM shared library that we'll load. If it isn't, then we won't need a path, because we'll be statically linking to the virtual machine.

```
#ifdef _WIN32
  Win32VmLoader loader( path, JNI_VERSION_1_2 );
#else
  UnixVmLoader loader( path, JNI_VERSION_1_2 );
#endif
```

If dynamic loading is turned on, then we use a Win32VmLoader for the Win32 platform. For now, we assume that all other platforms are Unix−like. In either case, we pass the user supplied path on to the loader.

```
#else
  StaticVmLoader loader( JNI_VERSION_1_2 );
#endif
```

In the case that we're doing static loading, we use the StaticVmLoader. We also need to make sure that we have our linker options set so that we are linking to the JVM library. There is no need to specify a path here.

```
OptionList options;

options.push_back( Classpath( "." ) );
options.push_back( Verbose( Verbose::Jni, Verbose::Class ) );
options.push_back( CustomOption( "−Xmx128M" ) );
```

Our choice of options isn't affected at all by the type of loading we perform. Here, we specify that we want a classpath set to the current directory, we want verbose logging for JNI and class loading, and we assume that we set the max heap to 128M (assuming that we're loading a Sun virtual machine, or some other virtual machine that supports this custom option).

```
try {
  jace::helper::createVm( loader, options );
}
catch ( std::exception& e ) {
  cout << "Unable to create the virtual machine: " << endl;
  cout << e.what();
  return −2;
}

cout << "The virtual machine was successfully loaded." << endl;
```

Finally, we create the virtual machine, specifying both the loader and the options.

Building and running

This example doesn't require the building of any proxies or peers. Just be careful to link to the JVM library if you want to use the StaticVmLoader or to globally #define JACE_WANT_DYNAMIC_LOADING if you want to use one of the dynamic VmLoaders.

Previous Next

Chapter 10
A Mapping Example

Using Maps

This example demonstrates the usage of a Java Map from C++. This example doesn't introduce a lot of new concepts. It just enforces the existing examples, and draws attention to a few interesting details. Like most of the other examples, this example, map_example only has a single source file, main.cpp, which contains the code we'll be examining:

```
include "jace/JNIHelper.h"
using jace::OptionList;

#include "jace/StaticVmLoader.h"
using jace::StaticVmLoader;

#include "jace/proxy/java/util/Set.h"
using jace::proxy::java::util::Set;

#include "jace/proxy/java/lang/System.h"
using jace::proxy::java::lang::System;

#include "jace/proxy/java/lang/Object.h"
using ::jace::proxy::java::lang::Object;

#include "jace/proxy/java/lang/Integer.h"
using jace::proxy::java::lang::Integer;

#include "jace/proxy/java/lang/String.h"
using jace::proxy::java::lang::String;

#include "jace/proxy/java/util/Map.h"
using jace::proxy::java::util::Map;

#include "jace/proxy/java/util/HashMap.h"
using jace::proxy::java::util::HashMap;

#include "jace/proxy/java/util/Map.Entry.h"
using jace::proxy::java::util::Map_Entry;

#include "jace/proxy/java/util/Iterator.h"
using jace::proxy::java::util::Iterator;

#include "jace/javacast.h"
using jace::java_cast;

#include <vector>
using std::vector;

#include <iostream>
using std::cout;
using std::endl;
```

By now, you should be familiar with all of these #includes. However, the include and using directive for Map_Entry should catch your attention. Map_Entry is actually an inner class, Entry, for the outer class java.util.Map. In Java notation, it is referred to as java.util.Map$Entry. When Jace generates the proxies for nested classes, it generates them as normal C++ classes. Because, '$' characters are illegal in C++ identifiers,

Jace translates all '$' characters to '_' characters. So, for example, a nested, nested class, FooBarBaz, would be generated as Foo_Bar_Baz. However, when naming the files, Jace uses '.' characters instead of '$' characters or '_' characters. This works well, because '$' characters are typically illegal in file names, and the use of '.' characters instead of '_' characters helps Jace to distinguish between an outer class named Foo_Bar and a nested class named Foo$Bar.

```
int main() {

  try {
    StaticVmLoader loader( JNI_VERSION_1_2 );
    jace::helper::createVm( loader, OptionList(), false );

    for ( int i = 0; i < 1000; ++i ) {
```

As before, we run this code in a loop to demonstrate that Jace manages references in all situations.

```
    Map map = HashMap();
```

Nothing new here. Following good coding style guidelines, we declare our variables to be of an interface type, rather than a concrete type.

```
    map.put( Integer( "1" ), String( "Hello 1" ) );
    map.put( Integer( "2" ), String( "Hello 2" ) );
    map.put( Integer( "3" ), String( "Hello 3" ) );
```

Here, we're just adding three entries to the Map. It's necessary to explicitly specify Integer and String, because there is no meaningful conversion from int or char* to jace::proxy::java::lang::Object − the argument types of Map.put.

```
    Set entrySet( map.entrySet() );

      for ( Iterator it( entrySet.iterator() ); it.hasNext(); ) {
        Map_Entry entry = jace::java_cast( it.next() );
        Integer key = jace::java_cast<Integer>( entry.getKey() );
        String value = jace::java_cast<String>( entry.getValue() );
        cout << "key: <" << key << "> value: <" << value << ">" << endl;
      }
```

We're just iterating through and printing out the Map's entrySet here. Normally, in Java code, you would type−cast from the return value of it.next() to Map_Entry, but you can't just use C style casts to perform casting of Java objects. Rather, to execute a Java 'type−cast' using Jace, you use the java_cast<> template function. You can use java_cast to cast between two Java types, or between a Java type and a JNI handle. If the cast fails, java_cast will throw a JNIException. You can also use the instanceof<> template function in the same way you'd use the instanceof operator in Java to determine if it is safe to perform a cast.

```
  }
  catch ( std::exception& e ) {
    cout << e.what() << endl;
    return −1;
  }

  return 0;
}
```

As always, we make sure to catch any exceptions that might occur.

Like the other examples, you can build this example by running ANT on build.xml. Other than having the JVM in your library path, there are no special requirements for running this example.

Chapter 11
Arrays in Action

Arrays and Iterators

This example demonstrates the usage of Java arrays from C++. Like most of the other examples, this example, array_example only has a single source file, array_example.cpp, which contains the code we'll be examining:

```
#define JACE_CHECK_ARRAYS
```

Jace has an optional array checking mechanism that you can turn on by #defining JACE_CHECK_ARRAYS in your code. This must be turned on or off for your entire project. When JACE_CHECK_ARRAYS is turned on, Jace checks for out of bounds indices and iterators.

```
#include "jace/JNIHelper.h"

#include "jace/proxy/types/JInt.h"
using jace::proxy::types::JInt;

#include "jace/proxy/java/lang/String.h"
using jace::proxy::java::lang::String;

#include "jace/proxy/java/lang/Integer.h"
using jace::proxy::java::lang::Integer;

#include "jace/proxy/java/lang/Object.h"
using jace::proxy::java::lang::Object;

#include "jace/JArray.h"
using jace::JArray;

#include "jace/StaticVmLoader.h"
using jace::StaticVmLoader;

#include "jace/OptionList.h"
using jace::OptionList;

#include "jace/JNIException.h"
using jace::JNIException;

#include <string>
using std::string;

#include <exception>
using std::exception;

#include <iostream>
using std::cout;
using std::endl;

#include <algorithm>
using std::for_each;

#include <functional>
using std::unary_function;
```

The new thing to notice here is the include of the algorithm and functional headers. Jace arrays can be used as

Standard C++ compliant containers with random access iterators.

```
struct print : public unary_function {
  void operator()( Object obj ) {
  cout << "[print] " << obj << endl;
  }
};
```

Here, we're just defining a functor that prints its parameter, for later use with for_each.

```
int main() {

  try {
    // Standard Vm setup
    StaticVmLoader loader( JNI_VERSION_1_2 );
    OptionList list;
    list.push_back( jace::CustomOption( "−Xcheck:jni" ) );
    list.push_back( jace::CustomOption( "−Xmx16M" ) );
    jace::helper::createVm( loader, list, false );

    typedef JArray StringArray;
```

Same old, same old along with a convenience typedef.

```
    // Creates a new array of Java String with 1000 null elements
    StringArray strArray( 1000 );

    // Fills the array with hello strings.
    for ( int i = 0; i < strArray.length(); ++i ) {
      strArray[ i ] = "Hello " + String::valueOf( JInt( i ) );
    }

    // Prints the contents of the array.
    // You can use JArray::operator[] for random access,
    for ( int j = 0; j < strArray.length(); ++j ) {
      cout << strArray[ j ] << endl;
    }
```

Whenever you create a new JArray, the array members get initialized the same way they do in Java, with the default value for the element type. Once we create the array, we assign each element a new String value. Note that we're using the assignment operator with the array index operator. Also note that we can retrieve the length of an array using JArray::length(). The length is cached, so it's not expensive to retrieve often.

```
    // Traverse the array again, but this time with JArray::Iterator
    // JArray::Iterator is preferred for non−random access,
    // because it allows Jace to perform smart caching.
    for ( StringArray::Iterator it = strArray.begin(); it != strArray.end(); ++it ) {
      *it = *it + " again";
      cout << *it << endl;
    }
```

JArray is similar to std::vector in that it allows random access through both an index operator and through a nested iterator class. JArray::Iterator is random access like vector::iterator, but it has some additional semantics attached to it. Because of the interface of an iterator, Jace can make some assumptions about caching that aren't as easy to make through an index operator. Users can also supply hints to the iterator to indicate, for example, how many elements are being iterated through. Therefore, the preferred means of accessing a JArray is through JArray::Iterator.

```
    // JArray::Iterator conforms to the Standard C++ Library's concept
    // of a random access iterator, and can be used that way.
    for_each( strArray.begin(), strArray.end(), print() );
```

A demonstration of the use of a JArray as a standard C++ container using a standard C++ algorithm.

```
    // Demonstrate some more random access iterator usage
    for ( StringArray::Iterator it2 = strArray.begin(); it2 <strArray.end() − 2; ) {
      it2 += 2;
      cout << "Forward two: " << *it2 << endl;
      it2 −= 1;
      cout << "Back one: " << *it2 << endl;
    }
```

Just some more random access iteration.

```
    // As stated above, Jace can check for bad array access
    // Here, we demonstrate some array access checking
    #ifdef JACE_CHECK_ARRAYS

      try {
        StringArray::Iterator it = strArray.begin( strArray.length() + 1 );
      }
      catch ( JNIException& e ) {
        cout << "Caught a bad iterator construction:" << endl;
        cout << e.what() << endl;
      }

      try {
        StringArray::Iterator it = strArray.end();
        ++it;
      }
      catch ( JNIException& e ) {
        cout << "Caught a bad iterator advancement." << endl;
        cout << e.what() << endl;
      }

      try {
        StringArray::Iterator it = strArray.begin();
        −−it;
      }
      catch ( JNIException& e ) {
        cout << "Caught a bad iterator rewind." << endl;
        cout << e.what() << endl;
      }

      try {
        cout << strArray[ strArray.length() ] << endl;
      }
      catch ( JNIException& e ) {
        cout << "Caught a bad array index." << endl;
        cout << e.what() << endl;
      }
    #endif
```

Here we actually demonstrate Jace's handling of bad array accesses, both through iterators and through the index operator.

```
 }
```

```
  catch ( exception& e ) {
  cout << e.what() << endl;
    return −1;
  }

  return 0;
}
```

Exception handling as usual.

Building and running

Like the other examples, you can build this example by running ANT on build.xml. Other than having the JVM in your library path, there are no special requirements for running this example.

Previous