

NewJ Library for C++ Developer's Guide



PureNative Software

www.pure-native.com

Copyright © 2002-2004 PureNative Software Corporation. All rights reserved.

PureNative, the PureNative logo, NewJ, the NewJ logo, NewJNI, NewJNIInterop, and Pie are trademarks of PureNative Software Corporation. Java is a registered trademark of Sun Microsystems, Inc. All other trademarks and registered trademarks are property of their respective owners.

The software described herein is licensed. Unauthorized reproduction or distribution of this software, or any portion of it, may result in severe civil and criminal penalties, and will be prosecuted to the maximum extent possible under law. Consult accompanying license agreement for additional permissions and restrictions.

NewJ Library for C++ is based on the Java 2 Platform, Standard Edition, version 1.2.2 API Specification from Sun Microsystems, Inc. *NewJNI* and *NewJNIInterop* are compatible with the JNI Specification from Sun Microsystems, Inc., and support any JNI-compatible version of the Java platform, including version 1.2, 1.3, or later. *NewJ Library for C++* is not a product of nor endorsed by Sun Microsystems, Inc.

Patent Pending.

Developer release 2: May 2004.

{5} NewJNI and NewJNIInterop Quick Tutorial

Important: Before you can begin the quick tutorial detailed in this chapter, you must generate C++ proxy classes for the version of the Java API you wish to use. Instructions are provided in the file “How to Generate Proxy Classes for Java API.html” in the “docs” directory. By default the fully-qualified directory is “C:\Program Files\newj0.6\docs”.

Naturally Integrating C++ and NewJ Library With External Java Classes and VMs

Even though it is possible to develop native applications entirely using C++ and NewJ Library, you may need to interoperate with the Java Platform itself. For instance, you may need to integrate existing compiled Java classes developed internally or provided by third-parties. Or, you may wish to leverage the capabilities of an installed Java Virtual Machine (JVM) and its associated implementation of the Java API. NewJ Library supports this and much more via its JNI Interoperability support. This support is provided by two related technologies, NewJNI and NewJNIInterop. First, let's consider why they are needed and how they fit into standard JNI.

Standard JNI: Capabilities and Limitations

Java Native Interface (JNI) is the standard technology for interfacing native applications and code with Java Virtual Machines (JVMs), especially the Java Runtime Environment (JRE) from Sun Microsystems. JNI was developed by Sun Microsystems and carries their recommendation and endorsement. It was designed primarily with C language development in mind, but it supports other languages as well, including C++.

Although JNI is a mature, widely-supported technology, by itself it suffers from certain limitations. Even though it is not ultra-difficult to learn, it can be cumbersome and challenging to use correctly. In this age of daily virus alerts and computer security warnings, JNI does not offer the level of safety and security expected of robust production systems. Indeed, many of its safety mechanisms are optional and implementation defined according to the JNI specification. JNI applications can be difficult to read, debug, and maintain further complicating matters. In fact, JNI applications bear little syntactic similarity to corresponding Java language applications. Furthermore, JNI does not fully leverage C++ to solve these problems. Specifically, it does not take advantage of C++ type safety and exception handling. Additionally, JNI does leverage the object-oriented features of C++, and it does not use C++ proxy classes for accessing the underlying Java classes. For these reasons a new solution is needed which leverages the standards compliance of JNI but fully embraces the usability of C++ coupled with modern techniques to ensure safe and secure programming.

NewJNI: The Natural C++ Interface to JNI

NewJNI is a new technology that builds on and extends industry standard JNI to meet the needs of industrial strength C++ development. NewJNI is 100% compatible with standard JNI, and its approach is completely platform-neutral. NewJNI includes the following features:

- Natural C++ language syntax.
- Full range of Java language features approximated in C++.
- Safer and more secure than standard JNI or ANSI/ISO C++ alone.
- Automatic object reference management and leak prevention.
- Highly efficient and very low overhead achieving full native performance.
- Supports multiple coexisting implementations of the Java API including cross-implementation method calls and integration.

NewJNIInterop: JNI Interoperability for NewJ Library

NewJNIInterop builds and extends both standard JNI and NewJNI in order to bring JNI Interoperability to NewJ Library, which is a 100% native C++ implementation of the core Java APIs. NewJNIInterop adds the following features beyond NewJNI alone:

- Includes separate 100% native C++ implementation of the core Java API for selective elimination of cross-environment calling overhead.
- Includes separate 100% native C++ approximation of Java language features for selective elimination of cross-environment calling overhead.
- Provides full interoperability between NewJ Library and JNI and between NewJ Library and NewJNI.

The NewJNI/NewJNIInterop Approach

NewJNI¹ and NewJNIInterop are solidly founded on standard JNI. NewJNI applications are real JNI applications and have full unfettered access to the underlying JNI functions and types. In fact, NewJNI supports and builds on all the standard JNI types, including primitive types, object reference types, and array reference types. The supported primitive types include: `jbyte`, `jchar`, `jshort`, `jint`, `jlong`, `jfloat`, `jdouble`, `jboolean`. The supported object reference types include: `jobject`, `jstring`. The supported array reference types include: `jbyteArray`, `jcharArray`, `jshortArray`, `jintArray`, `jlongArray`, `jfloatArray`, `jdoubleArray`, `jbooleanArray`, `jobjectArray`. NewJNI defines numerous C++ classes and methods which make using JNI and its standard types easier and safer.

The NewJNI classes make use of the Proxy design pattern. In essence, the NewJNI proxy classes wrap Java classes and methods in real C++ objects. These C++ objects look and feel like the original Java classes they are wrapping. When you make a new instance of one of

¹For brevity, the term NewJNI will be used to apply to both NewJNI and NewJNIInterop unless otherwise indicated.

these C++ proxy classes a corresponding instance of the Java class is also created. When you call a method on a C++ proxy class, it will call through to the underlying Java class. There is even a natural C++ interface for accessing object fields, whether they are instance fields or static fields.

NewJNIInterop utilizes the Adapter design pattern to accomplish JNI interoperability. NewJNIInterop adapts both NewJNI classes and JNI functions to provide a logical interface between NewJ Library and JNI. NewJNIInterop consists of the following major classes: NewJNIInteropSystem, NewJNIEnvInterop. NewJNIInteropSystem extends NewJNISystem, and NewJNIEnvInterop subclasses NewJNIEnv. These classes coupled with NewJNI make it easy and straightforward to develop NewJ applications that take advantage of JNI interoperability.

NewJNI and NewJNIInterop consist of software libraries and tools. The NewJNI and NewJNIInterop runtime classes are packaged together in the newjinterop library. A complete collection of proxy classes for the Java 2 Standard Edition (J2SE) API is provided in the newjni2 library. You may use the newjni2 library provided or you may build your own instead. NewJNI includes the NJNIGen tool for generating C++ proxy classes from compiled Java classes, be they standard Java API classes, third-party classes, or those that you have developed yourself. The NJNIGen tool is a Java application that may be run on any Java 2 VM. Furthermore, several NewJ AppWizards are included for Visual C++ 6.0 which automate the process of creating NewJ applications that support JNI interoperability. (For more information on the NewJ AppWizards, see *Chapter 6: Adding NewJ/NewJNI/NewJNIInterop Support to Your Application*.) The applications generated by the wizards are automatically configured to link to the newjninterop and newjni2 libraries.

NewJNI Quick Tutorial

This brief tutorial teaches you the basics of NewJNI by example source code. So as not to overwhelm you, every detail of the example source code not be explained at once. Instead, it will be explained progressively and in order of importance. This tutorial if followed by in-depth sections covering all the intricate details. If you can't wait for that, feel free to jump ahead.

NewJNI Quick Tutorial: Creating Object Instances

First, let's create a new object instance and call a method on it. In this example, we'll create an instance of the popular Vector class with an initial capacity of 500 elements. In order to instantiate the Vector class, we must include the definition of its C++ proxy class. This is necessary because C++ does not automatically include referenced class definitions unlike the Java language which does so automatically. This is done with NewJNI as follows:

```
#include <newjni/newjni.h>
#include <newjni/java/util/Vector.h>
```

For other Java packages and classes, simply change “java/util/Vector” to the desired class. Note that “newjni” is the root namespace of all C++ proxy objects for the standard Java API.

Did you note the inclusion of “newjni/newjni.h”. This is the base header for all NewJNI applications and must be included before any other NewJNI headers or proxy classes.

Now let's instantiate the object. In the Java language we would instantiate the object as follows:

```
java.util.Vector myVector = new java.util.Vector(500);
```

The NewJNI version is very similar. Here's how to do it with NewJNI:

```
::newjni::java::util::VectorR myVector = new ::newjni::java::util::Vector(500);
```

That's it. There's now a living, breathing, VM-created instance of Vector underneath. In fact, since it was created by JNI, there's even a jobject beneath it all, which is a global object reference.

Did you notice some of the differences between the two versions? Again, with NewJNI all standard Java API packages and classes are contained in the root newjni namespace, hence “::newjni:” prefixed to “java::util::Vector”. Also, double semicolons (::) are used as namespace and class name separators instead of the period (.) used in the Java language. Lastly, C++ makes a clear distinction between reference types and the actual types they are referencing. NewJNI reference types, which are akin to highly sophisticated “smart pointers,” act just like object references in the Java language. By convention, NewJNI reference types always end with a single capital R. In this example, VectorR is the reference type and Vector is the type being referenced. How do you know when to use which form. Put simply, reference types are assigned to and are generally left-hand side arguments, whereas the types being referenced are usually instantiable or castable and are generally right-hand side arguments.

NewJNI Quick Tutorial: Calling Instance Methods

Next, let us call the capacity() method of our newly created instance to make sure it's actually 500 as we specified. This may be done as follows in the Java language:

```
int myCapacity = myVector.capacity();
assert(myCapacity == 500);
```

And here's the NewJNI version:

```
jint myCapacity = myVector->capacity();
```

```
pi_assert(myCapacity == 500);
```

Note the use of the standard JNI type jint for int in the Java language. NewJNI follows the pattern of using standard JNI types wherever possible. Notice too that instead of separating variable names from members fields and methods with a period (.) like in the Java language, NewJNI uses the -> operator instead just like many smart pointers do.

pi_assert() is part of the NewJNI assertion facility. Like the assert() statement introduced in Java 1.4, pi_assert() takes a single boolean expression. If it evaluates to false, an assertion message is displayed. By default, NewJNI assertions are only enabled in debug builds not release builds. In release builds, they are excluded and incur no overhead whatsoever.

NewJNI Quick Tutorial: Accessing and Assigning Member Fields

How about printing the capacity to standard “out”? Good question. The answer will demonstrate how to access member fields like “out” in NewJNI. First, let's see the usual way to do this in the Java language.

```
java.lang.System.out.println(“The capacity is...”);  
java.lang.System.out.println(capacity);
```

This same thing may be accomplished with NewJNI as follows:

```
#include <newjni/java/lang/System.h>  
  
...  
  
::newjni::java::lang::System::out_()->println(PI_JT(“The capacity is...”));  
::newjni::java::lang::System::out_()->println(capacity);
```

Note that member fields are treated as methods. In fact, these special methods are called “field accessor methods.” Why is this approach taken? Two reasons. First, it guarantees that field values are obtained automatically and on demand from the underlying VM object instance. This increases performance and reliability. Second, the Java language allows the same identifier name to be used for both a field and a method in the same class definition but C++ does not. By turning such fields into methods and appending an underscore to such names, it is possible to support such like-named fields and methods even in standard ANSI/ISO C++.

If you can access fields using NewJNI, can you set them just as easily? The answer is yes. To illustrate, here is a sample in the Java language:

```
java.awt.Point myPoint = new java.awt.Point();  
myPoint.x = 1280;  
myPoint.y = 1024;  
assert(myPoint.x == 1280);  
assert(myPoint.y == 1024);
```

This is how to accomplish the same thing with NewJNI in C++:

```
#include <newjni/java/awt/Point.h>

...

::newjni::java::awt::PointR myPoint = new ::newjni::java::awt::Point();
myPoint->x_() = 1280;
myPoint->y_() = 1024;
pi_assert(myPoint->x_() == 1280);
pi_assert(myPoint->y_() == 1024);
```

NewJNI Quick Tutorial: Object Identity

Object identity is an important part of Java programming. The Java language provides the `instanceof` keyword to test if a certain object reference is-a certain class. NewJNI provides the same feature via the `pi_instanceof` and `pi_abstractinstanceof` macros or simulated keywords. `pi_instanceof` is for checking against non-abstract classes, whereas `pi_abstractinstanceof` is for checking against abstract classes and interfaces. Here the syntax of both:

```
pi_instanceof(objectAddress, nonAbstractClassName)

pi_abstractinstanceof(objectAddress, abstractClassName)
```

where *objectAddress* is the address of an object instance, *nonAbstractClass* is a non-abstract class name, and *abstractClassName* is an abstract class or interface name.

Here is an example of using `pi_instanceof` and `pi_abstractinstanceof`:

```
::newjni::java::awt::PointR myPoint = new ::newjni::java::awt::Point();
if (pi_instanceof(*myPoint, ::newjni::java::awt::Rectangle))
{
    ::newjni::java::lang::System::out_()->println(
        PI_JT("Point is a Rectangle"));
}
else
{
    ::newjni::java::lang::System::out_()->println(
        PI_JT("Point is not a Rectangle"));
}
```

```

if (pi_jabstractinstanceof(*myPoint, ::newjni::java::io::Serializable))
    {
        ::newjni::java::lang::System::out_()->println(
            PI_JT("Point is Serializable"));
    }
else
    {
        ::newjni::java::lang::System::out_()->println(
            PI_JT("Point is not Serializable"));
    }

```

NewJNI Quick Tutorial: Casting

Now that you are familiar with how NewJNI supports checking object identity, you may be wondering, “What about casting?” NewJNI includes several macro for casting. They are:

```

pi_jcast(nonAbstractClassName, objectReference)

pi_jabstractcast(abstractClassName, objectReference)

```

where *nonAbstractClassName* is a class name, *abstractClassName* is an abstract class or interface name, and *objectReference* is an instance of a NewJNI proxy reference type.

For example,

```

::newjni::java::awt::PointR myPoint = new ::newjni::java::awt::Point();
::newjni::java::lang::ObjectR myObject = myPoint;
myPoint = pi_null; // force myPoint reference to be released
myPoint = pi_jcast(::newjni::java::awt::Point, myObject);

```

Did you notice that myPoint is assigned to pi_null? pi_null is actually a special NewJNI type that promotes safety and correct usage. In order to assign an instance of a reference type to null, pi_null must be used instead of other common values, such as NULL or 0, which are not permitted.

NewJNI Quick Tutorial: Catching Exceptions

The foregoing is an example of a valid cast. What about an invalid cast? Just like the Java language, invalid casts throw the ClassCastException. In NewJNI, actually an instance of the ClassCastException reference type is thrown. Here's an example of how to catch it:

```

::newjni::java::awt::PointR myPoint = new ::newjni::java::awt::Point();
try
    {
        ::newjni::java::awt::RectangleR myRectangle =
pi_jcast(::newjni::java::awt::RectangleR, myPoint);
    }

```

```

    }
    catch (::newjni::java::lang::ClassCastExceptionR& exception)
    {
        ::newjni::java::lang::System::out_()->println(exception->getMessage());
    }
}

```

NewJNI Quick Tutorial: Creating Arrays

NewJNI provides a simplified, natural way to create arrays. This feature is provided through the `pi_newarray` macro. Here's its syntax:

```
pi_newarray(type, elementCount)
```

where *type* is any JNI fundamental type (which excludes object and its derivatives) or any NewJNI proxy class name, and *elementCount* is the count of elements to allot for the new array. The NewJNI runtime automatically initializes the elements to the values mandated by the Java language.

Here is an example of making a new integer array of 15 elements:

```
pi_jintAR anIntArray = pi_newarray(pi_jint, 15);
```

To explain, `pi_jintAR` is the NewJNI proxy array class for integer arrays. All other proxy classes for fundamental types follow the same naming pattern. “pi_” is followed by the JNI type name, which itself is followed by “AR” for array reference. Additionally, `pi_newarray` requires that “pi_” must be prefixed to the JNI type name.

NewJNI Quick Tutorial: Accessing and Assigning Array Elements

Accessing and assigning array elements of arrays is intuitive and natural with NewJNI. In fact, it is almost identical to how you would do so in the Java language. For example, here is how you would access the seventh element of the above an array of integers named `anIntArray` using NewJNI:

```
anIntArray[5]
```

That's it. No different from the Java language. How about assignment? To assign the above element to the value 64870, you would use the following statement:

```
anIntArray[5] = 64870;
```

Looks just like the Java language, doesn't it? Behind the scenes, this statement also automatically updates the underlying VM-based array element as well. It all works, all without any complicated or error-prone JNI programming.

NewJNI Quick Tutorial: Working With Namespaces

In the preceding examples, class names have been fully qualified with their namespaces, C++'s approximation of the Java language's packages. And such fully qualified namespaces can be wearisome and cumbersome to work with. In light of this, we'll learn how to use shorter, unqualified class names in your code where appropriate.

In the Java language, the `import` keyword takes care of this feature. It takes the following form:

```
import packageName;  
import fullyQualifiedClassName;
```

where *package name* is the name of a package, such as `java.io`, and *fullyQualifiedClassName* is the fully-qualified name of a class, such as `java.awt.Frame`.

To import a package name, or namespace, in NewJNI, adhere to the following form:

```
using namespace theNamespace;
```

where *theNamespace* is the NewJNI namespace to use, such as `::newjni::java::io`.

To import a class name in NewJNI, apply the following form:

```
using fullyQualifiedClassName;
```

where *fullyQualifiedClassName* is the fully qualified NewJNI class name, such as `::newjni::java::awt::Frame`.

The above statement may not do everything you're expecting it to do. Like what? It does not import the NewJNI reference type (which ends with a capital R) for the specified class name or its array reference types (which end with a capital AR and capital AAR, respectively). To import these, apply this from:

```
using fullyQualifiedClassNameR; // the reference type  
using fullyQualifiedClassNameAR; // the 1D array reference type  
using fullyQualifiedClassNameAAR; // the 2D array reference type
```

And here's an example for the `Frame` class:

```
using ::newjni::java::awt::FrameR; // the reference type  
using ::newjni::java::awt::FrameAR; // the 1D array reference type  
using ::newjni::java::awt::FrameAAR; // the 2D array reference type
```

Now all the reference types and array reference types have also been imported. You may be thinking, "That was a lot of error-prone typing after all and wasn't our original goal to cut down on verbosity and errors? There's got to be a better way." And you would be right. The `pi_importclass` macro is provided to do all that. Here's its syntax:

```
pi_importclass(fullyQualifiedClassName);
```

Died-in-wool C++ purists may prefer the expanded form, but you may opt for the macro due to its simplicity and terseness.

Finally, you should be aware that there are some caveats to the using keyword (and the `pi_importclass` macro). In fact, you should be very careful with the using keyword (and the `pi_importclass` macro) because it can have unwelcome side effects if it is used improperly. Here are some general guidelines for your benefit:

- Do not put using statements inside header files.
- Always put using statements after all `#include` statements. Otherwise, the header files that are included may be adversely affected by the using statements.
- Do not put using statements inside namespaces. Otherwise, the namespaces themselves may be adversely affected by the using statements.

If you follow these guidelines, you will avoid the pitfalls of improperly applying the using statement, and instead you will find it much easier to work with namespaces.

This completes the quick tutorial. Now you're ready to delve into the nitty gritty of NewJNI, its core concepts.