

# NewJ Library for C++ Developer's Guide



**PureNative** Software

[www.pure-native.com](http://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.

# {D} NewJNI and NewJNIInterop Reference

## NewJNI Core Concepts

NewJNI leverages the core concepts of JNI and Pie Run-time Library. (To familiarize yourself with the concepts of Pie Run-time Library, see *Chapter 4: Pie Run-time Library Reference*.) NewJNI provides access to VM-based Java language features and constructs in natural ANSI/ISO C++. NewJNI accomplishes this by defining macros, classes, and templates, which approximate or correspond to Java language features and constructs. NewJNI applications closely resemble Java language application in syntax and form of expression. This resemblance makes it possible to convert, or port, Java language source code to C++ or to develop new C++ applications which take advantage of Java language features.

## Fundamental Classes

NewJNI includes the following fundamental classes:

- **NewJNISystem.** Manages initialization and termination of the NewJNI environment as well as the underlying JNI environment. This class also provides helper functions that simplify JNI programming.
- **NewJNIEnv.** Natural C++ adaption of the standard JNIEnv type used throughout JNI programming. This class does the bulk of the work in C++ proxy classes.
- **NewJNIObject.** Base class of all C++ proxy classes. This class also provides support for automatic object reference management.

## Macros for Approximating Java Keywords

NewJNI defines macros (also called “simulated keywords”) for approximating Java’s built-in keywords. Here is a list of NewJNI macros and their corresponding keywords in Java.

## Simulated Keywords: Class Declaration Modifiers

NewJNI support the following Pie macros, which are called “class declaration modifiers.” They are used in class declarations. They must be preceded by the C++ class keyword, as in,

```
class pi_interfaceclass MyClass
```

<b>Pie Macro (Simulated Keyword)</b>	<b>Java Keyword</b>
pi_interfaceclass	Interface
pi_abstractclass	abstract (as used in “abstract class”)
pi_finalclass	final (as used in “final class”)
pi_publicclass	public (as used in “public class”)
pi_privateclass	private (as used in “private class”)

### **Simulated Keywords: Method and/or Field Modifiers**

NewJNI supports the following Pie macros, which are used when declaring methods and fields as part of a class. They are called “method modifiers” and “field modifiers,” respectively.

<b>Pie Macro (Simulated Keyword)</b>	<b>Java Keyword</b>
pi_protected	protected
pi_packageprivate	N/A (default method/field access modifier in Java)
pi_native	native
pi_transient	transient
pi_abstract	abstract (as used in a method declaration)
pi_finalmethod	final (as used in a method declaration)
pi_finalfield	final (as used in a field declaration)
pi_staticfield	static (as used in a field declaration)

### **Simulated Keywords: Exception Specification**

NewJNI supports the following Pie macros, which are used for specifying exceptions that may be thrown by a particular constructor or method.

<b>Pie Macro (Simulated Keyword)</b>	<b>Java Keyword</b>
pi_throws	throws (followed by only one class name)
pi_throwsbegin	throws (followed by one or more class names)
pi_throwsalso	N/A (unnecessary in Java)
pi_throwsend	N/A (unnecessary in Java)

## Simulated Keywords: Object Identity and Management

The following NewJNI macros are used for managing object identity and synchronization.

NewJNI Macro (Simulated Keyword)	Java Keyword
pi_null	null
pi_jinstanceof	instanceof (as used with a non-abstract class)
pi_jabstractinstanceof	instanceof (as used with an abstract class or interface)
pi_jcast, pi_jabstractcast, pi_jobjectadapt, pi_jobjectadopt	N/A (casting is part of Java language syntax)

## Simulated Keywords: Assertion Facility<sup>5</sup>

NewJNI supports the following Pie macros in assertion statements.

Pie Macro (Simulated Keyword)	Java Keyword
pi_assert	assert (as used with a single argument)
pi_assert2	assert (as used with two arguments)

Additional NewJNI macros used for object and array creation are detailed later in this chapter.

---

<sup>5</sup> Assertions became a standard Java language feature starting with version 1.4. If you require strict compatibility with Java 1.2.2 language features, assertions should not be used in Java language source code. However, the Pie assertion facility used in NewJNI/C++ is a 100% native C++ implementation, so it may be used freely with any version of Java (including versions before 1.4) without affecting Java binary compatibility in any way.

## Fundamental Types

NewJNI uses the standard fundamental types defined by JNI. Here is a list of JNI fundamental types and their corresponding fundamental types in the Java language.

JNI Fundamental Type	Java Fundamental Type
jboolean	boolean
jbyte	byte
jchar	char
jshort	short
jint	int
jlong	long
jfloat	float
jdouble	double

For the sake of source code compatibility with existing JNI source code, the JNI jboolean type is supported over the standard C++ bool type. However, where source code compatibility this is not at issue, the standard C++ bool type is used.

## Object Reference Types

Lifetimes of object instances are managed by NewJNI reference types. NewJNI reference types use thread-safe reference counting to keep objects alive. When a reference type instance is set to an object instance, the reference count for the object instance is incremented by 1. When a reference type instance no longer refers to a particular object instance, the reference count for that particular object instance is decremented by 1. When the reference count reaches zero the object instance is deleted.

Reference types must be explicitly defined for each class and interface defined. In addition, reference types must be defined for a particular class or interface in order use Pie macros with that particular class or interface.

To define reference types for a class, use one of the `NEWJNI_DEFINE_DERIVED_REFERENCES` macro. If your class does not implement any interfaces in its class definition use the `NEWJNI_DEFINE_DERIVED_REFERENCES`. On the other hand, if your class implements one or interfaces in its class definition use the macro named `NEWJNI_DEFINE_DERIVED_REFERENCES` followed by the number of interfaces your class implements. For instance, if your class implements three interfaces use `NEWJNI_DEFINE_DERIVED_REFERENCES3`.

The `NEWJNI_DEFINE_DERIVED_REFERENCES` is declared as follows:

```
NEWJNI_DEFINE_DERIVED_REFERENCES(className,superclassName);
```

where *className* is the name of your class, and *superclassName* is the name of the your superclass. (Just like Java, all NewJNI classes have an explicit superclass except for the base Object class. All NewJNI classes derive from Object or from another class which descends from Object.) For example,

```
NEWJNI_DEFINE_DERIVED_REFERENCES(MyDerivedClass, Object);
```

The other `NEWJNI_DEFINE_DERIVED_REFERENCES` macros are declared as follows:

```
NEWJNI_DEFINE_DERIVED_REFERENCES $n$ (className,superclassName,...);
```

where  $n$  is the number of interfaces your class implements (1 or more), *className* is the name of your class, *superclassName* is the name of the your superclass, which is then followed by the name of each interface your class implements. (All NewJNI classes have a superclass except for the Object class. All NewJNI classes derive from Object or from another class which descends from Object.) For example,

```
NEWJNI_DEFINE_DERIVED_REFERENCES(MyDerivedClass, Object);
```

To define reference types for an interface, use either the `NEWJNI_DEFINE_INTERFACE_REFERENCES` or `NEWJNI_DEFINE_DERIVED_INTERFACE_REFERENCES` macros. If your interface is a base interface and does not extend another interface use `NEWJNI_DEFINE_INTERFACE_REFERENCES`. Otherwise, use `NEWJNI_DEFINE_DERIVED_INTERFACE_REFERENCES`.

The `NEWJNI_DEFINE_INTERFACE_REFERENCES` is declared as follows:

```
NEWJNI_DEFINE_INTERFACE_REFERENCES(interfaceName);
```

where *interfaceName* is the name of your interface. For example,

```
NEWJNI_DEFINE_INTERFACE_REFERENCES(MyBaseInterface);
```

The `NEWJNI_DEFINE_DERIVED_INTERFACE_REFERENCES` is declared as follows:

```
NEWJNI_DEFINE_DERIVED_INTERFACE_REFERENCES(interfaceName,superinterfaceName);
```

where *interfaceName* is the name of your interface, and *superinterfaceName* is the name of the interface you are extending. For example,

```
NEWJNI_DEFINE_DERIVED_INTERFACE_REFERENCES(MyDerivedInterface, MyBaseInterface);
```

Reference types are named based on the class or interface they reference followed by the capital letter R. For example, the reference type for MyClass would be named MyClassR. The reference type name is generally only used when declaring reference type fields or variables. For example,

```
MyClassR myClass;

void MyClass::myMethod(MyClassR myClassArgument)
{
    ...
}
```

On the other hand, the reference type name is generally not used with the NewJNI macros but the actual class name is. This is demonstrated under the next subheading.

## Object Creation

New object instances may be created naturally using the new keyword as follows:

```
new constructor
```

where *constructor* is an actual constructor name along with any accompanying arguments. For example,

```
new Integer(847)
```

The only caveat to the above form of object creation is that it is susceptible to resource leaks if the new instance is not properly assigned to a reference type which is not automatically performed in C++ although it is in the Java language. To address this safety concern, the `pi_new` macro is provided. The `pi_new` macro can take the place of the new keyword and is declared as follows:

```
pi_new(object, constructor)
```

where *object* is the type (class name) of object instance being created and *constructor* is an actual constructor name along with any accompanying arguments. For example,

```
pi_new(Integer, Integer(847))
```

There is the additional requirement that the specified type must define corresponding reference types. This is easily satisfied by using one of the `NEWJNI_DEFINE_DERIVED_REFERENCES` macros.

Object instances created using `pi_new` are automatically adopted by a hidden anonymous reference. This prevents inadvertently orphaning the newly created object and causing a resource leak.

## String Literals

NewJNI treats strings as full-fledged objects just like Java does. In C++, string literals are character arrays by nature instead of real objects. To make a string literal into a string object, the `PI_JT` macro is provided, which is similar in form to certain string literal macros used in other C++ libraries. It is declared as follows:

```
PI_JT(stringLiteral)
```

Here is an example:

```
StringR sampleString = PI_JT("Sample String");
```

In Java, there can be only one instance of every unique string literal. Even though the same string literal may appear more than once in a program, internally only one copy of the string literal is maintained. NewJNI supports this same notion of string literals. This makes it possible to compare two string literals by identity instead of relying on the `equals()` method. Consider this example.

```
StringR firstString = PI_JT("String literal");

StringR secondString = PI_JT("String literal");

if (firstString->equals(secondString))
    {
        System.out.println("The two strings are equal.");
    }

if (firstString->_isSameObject(secondString))
    {
        System.out.println("The two strings are the same object.");
    }
```

This fragment will output the following:

```
The two strings are equal.
The two strings are the same object.
```

The output of the second line indicates that NewJNI properly treats string literals the same as the Java language does. This is a fundamental feature of the Java language on which many programs depend. As a side benefit, it lowers memory requirements for storing string literals and increases the performance of certain string comparison operations because comparing the identity of two objects is much faster than using the `equals()` method.

The `PI_JT` macro is suitable for all string literals except for those that contain Unicode escape sequences (“`\udddd`”). The `PI_JTE` macro is provided in order to support Unicode escape sequences in NewJNI. (The “E” in `PI_JTE` stands for “escape [sequence].”) It is declared as:

```
PI_JTE(stringLiteral)
```

where *stringLiteral* is a string literal, which may contain one or more *modified* Unicode escape sequences. (The modified form is necessary because C++ compilers generally do not support Unicode escape sequences.) The modified form taken by the `PI_JTE` macro is “`\udddd`”, which differs slightly from the form “`\u`” employed in the Java language. Due to this form of Unicode escape sequences in NewJNI, the normal “`\\u`” sequence must instead be encoded as “`\\\\u`”. This latter requirement does not apply to `PI_JT` macro, though.

Here is an example of a Unicode escape sequence in the Java language:

```
String sequence = “Sample characters \u0b87 \u2603 \\u”;
```

And here it is using the `PI_JTE` macro in C++:

```
StringR sequence = PI_JTE(“Sample characters \\u0b87 \\u2603 \\\\u”);
```

## Character Literals

The Java language treats all character literals as 16-bit Unicode characters and also allows character literals to be expressed as Unicode escape sequences. On the other hand, ANSI C++ does not guarantee 16-bit Unicode support because the wide character type (`wchar_t`) is implementation dependent and may vary from one compiler or platform to the next. Even if `wchar_t` is a 16-bit type, ANSI C++ still does not support 16-bit Unicode escape sequences. In order to guarantee that character literals are properly treated as Unicode, you must put each character literal within the `PI_C()` macro. It takes this form:

```
PI_C(c)
```

where *c* is a single character.

Here is an example:

```
PI_C('A')
```

When working with `String` objects, it is recommended that you use `PI_C()` instead of `PI_JT()` whenever you need to append a single character because `PI_C()` is much more efficient.

You may wonder, Does `PI_C()` take care of 16-bit Unicode escape sequences? The answer is no. Instead, the `PI_CU()` macro is provided for this purpose. It follows the form:

PI\_CU(*bbbb*)

where *b* is a hexadecimal digit (0-F).

For example,

PI\_CU(2F98)

## Arrays

NewJNI supports arrays of fundamental types, arrays of objects, and arrays of interfaces. Like the Java language, arrays are in fact objects themselves and their lifetimes are managed by reference types. The following subheadings detail reference types and array creation for both fundamental type arrays and object arrays.

### Fundamental Type Arrays: Reference Types

Array reference types are automatically defined for all fundamental types.

Array reference type names are prefixed with “pi\_” followed by the name of the fundamental type they reference and finally the capital letters AR. For example, the reference type for jint (int in Java) would be named pi\_jintAR. The array reference type name is generally only used when declaring array reference type fields or variables. For example,

```
pi_jintAR myIntArray;  
  
void MyClass::myMethod(pi_jintAR myIntArrayArgument)  
{  
    ...  
}
```

On the other hand, the array reference type name is generally not used with the NewJNI macros but the actual type name is. This is demonstrated later in this chapter.

### Object Arrays: Reference Types

Array reference types must be defined for all classes you define. This is easily satisfied for classes by using one of the NEWJNI\_DEFINE\_DERIVED\_REFERENCES macros; these macros define both object reference types and array reference types without any extra work on your part.

Array reference types are named based on the name of the class they reference followed by the capital letters AR. For example, the array reference type for MyClass would be named

MyClassAR. The array reference type name is generally only used when declaring array reference type fields or variables. For example,

```
MyClassAR myClassArray;  
  
void MyClass::myMethod(MyClassAR myClassArrayArgument)  
{  
    ...  
}
```

On the other hand, the array reference type name is generally not used with the NewJNI macros but the actual class name is. This is demonstrated under the next subheading.

### Creating One-Dimensional Arrays

New one-dimensional arrays may be created by using the `pi_newarray` macro. The `pi_newarray` macro is declared as follows:

```
pi_newarray(typeName, length)
```

where *typeName* is the name of the fundamental type or name of the class contained in the array, and *length* is the number of the elements in the array. Here is an example creating a fundamental type array:

```
pi_newarray(pi_int, 10)
```

Here is an example creating an object array:

```
pi_newarray(StringBuffer, 14)
```

There is the additional requirement that the specified type must define corresponding reference types. This is easily satisfied for classes by using one of the `NEWJNI_DEFINE_DERIVED_REFERENCES` macros. As mentioned in an earlier subheading, this is unnecessary for fundamental types because array reference types are automatically defined for all fundamental types.

Arrays created using `pi_newarray` are automatically adopted by a hidden anonymous array reference. This prevents inadvertently orphaning the newly created array and causing a resource leak.

Newly created one-dimensional arrays are also automatically initialized to their default values. To specify initialization values for arrays, see the later section “One-Dimensional Array Initialization.”

### Creating Two-Dimensional Arrays

New two-dimensional arrays may be created by using the `pi_newarray2d` macro. The `pi_newarray2d` macro is declared as follows:

```
pi_newarray2d(typeName,d1,d2)
```

where *typeName* is the name of the fundamental type or name of the class or interface contained in the array, *d1* is the number of elements in the array's outer dimension, and *d2* is the number of elements in the array's inner dimension. Here is an example creating a two-dimensional fundamental type array:

```
pi_newarray2d(pi_int, 3, 5)
```

Here is an example of creating a two-dimensional String array:

```
pi_newarray2d(String, 2, 10)
```

For non-fundamental types, that is, classes and interfaces, there is the additional requirement that the specified type must define corresponding reference types. This is easily satisfied for classes by using one of the following macros:

```
NEWJNI_DEFINE_DERIVED_REFERENCES,  
NEWJNI_DEFINE_INTERFACE_REFERENCES,  
NEWJNI_DEFINE_DERIVED_INTERFACE_REFERENCES.
```

For details about these macros, see the earlier section “Object Reference Types.”

Arrays created using `pi_newarray2d` are automatically adopted by a hidden anonymous array reference. This prevents inadvertently orphaning the newly created array and causing a resource leak.

Newly created two-dimensional arrays are also automatically initialized to their default values. To specify initialization values for them, see the later section “Two-Dimensional Array Initialization.”

## One-Dimensional Array Initialization

Initial values for one-dimensional array elements may be provided using the `NEWJNI_BEGIN_ARRAY_VALUES` and `PIE_END_ARRAY_VALUES` macros. These macros also take care of array creation so their use eliminates the need for `pi_newarray` as well. `NEWJNI_BEGIN_ARRAY_VALUES` takes the following form:

```
NEWJNI_BEGIN_ARRAY_VALUES(typeName, length, lengthToInitialize)
```

where *typeName* is the name of the fundamental type or name of the class or interface contained in the array, *length* is the number of the elements in the array, and *lengthToInitialize* is the number of elements in the array that are being initialized. *lengthToInitialize* is usually the same as *length*, however it may be less than *length*. *lengthToInitialize* may never be more than *length*. Both *length* and *lengthToInitialize* must be greater than zero, otherwise you must use `pi_newarray` instead.

NEWJNI\_END\_ARRAY\_VALUES takes this simple form:

```
NEWJNI_END_ARRAY_VALUES()
```

Between NEWJNI\_BEGIN\_ARRAY\_VALUES and NEWJNI\_END\_ARRAY\_VALUES provide the values you wish to initialize your array elements to. Here is an example of initializing a fundamental type array:

```
pi_jintAR myIntArray =
    NEWJNI_BEGIN_ARRAY_VALUES(pi_int, 8, 3)
        21,
        45,
        100
    NEWJNI_END_ARRAY_VALUES();
```

This will create a new integer array that has eight elements. The first three elements will be initialized to 21, 45, and 100, respectively. The succeeding five elements are automatically initialized to their default values.

## Two-Dimensional Array Initialization

Initial values for two-dimensional array elements may be provided using the NEWJNI\_BEGIN\_ARRAY\_2D\_VALUES and NEWJNI\_END\_ARRAY\_2D\_VALUES macros. These macros also take care of array creation so their use eliminates the need for pi\_newarray2d as well. NEWJNI\_BEGIN\_ARRAY\_2D\_VALUES takes the following form:

```
NEWJNI_BEGIN_ARRAY_2D_VALUES(typeName, length, lengthToInitialize)
```

where *typeName* is the name of the fundamental type or name of the class or interface contained in the array, *length* is the number of the elements in the array, and *lengthToInitialize* is the number of elements in the array that are being initialized. *lengthToInitialize* is usually the same as *length*, however it may be less than *length*. *lengthToInitialize* may never be more than *length*. Both *length* and *lengthToInitialize* must be greater than zero, otherwise you must use pi\_newarray2d instead. If *lengthToInitialize* is less than *length*, the remaining outer-level elements will automatically be initialized to null.

NEWJNI\_END\_ARRAY\_VALUES takes this simple form:

```
NEWJNI_END_ARRAY_2D_VALUES()
```

Between NEWJNI\_BEGIN\_ARRAY\_2D\_VALUES and NEWJNI\_END\_ARRAY\_2D\_VALUES provide the values you wish to initialize your outer-level array elements to. These values may be one-dimensional arrays or pi\_null as the case may be. Here is an example of initializing an array of arrays of String objects. (This example assumes that the java::lang namespace is being used.)

```
StringAAR anArrayOfStringArrays =
    NEWJNI_BEGIN_ARRAY_2D_VALUES(String, 3, 3)
        NEWJNI_BEGIN_ARRAY_VALUES(String, 2, 2)
            PI_JT("January"),
            PI_JT("February")
        NEWJNI_END_ARRAY_VALUES(),
    pi_null,
    NEWJNI_BEGIN_ARRAY_VALUES(String, 4, 3)
        PI_JT("Monday"),
        PI_JT("Tuesday"),
        PI_JT("Wednesday")
    NEWJNI_END_ARRAY_VALUES()
NEWJNI_END_ARRAY_2D_VALUES();
```

This will create a new two-dimensional String array with three outer-level elements. Interestingly, it is not a matrix because each outer-level element does not have the same number of inner elements. The first element contains a String array with two elements, "January" and "February". The second element is null and doesn't contain an array at all, which is completely acceptable. The third element contains a String array of four elements, only three of which are explicitly initialized, "Monday", "Tuesday", and "Wednesday".

## Classes

In the Java language classes are declared and defined in a single class definition block. Unlike the Java language, classes in C++ are generally declared in one (source) unit and defined in a separate (source) unit. The following methods describe how to declare and define classes.

**Note:** Subclassing NewJNI C++ proxy classes is primarily intended for making new C++ proxy classes for a correspondingly derived VM-based class. For the NewJNI String class derives from the NewJNI Object class in order to correspond to the relationship between the VM-based [java.lang.String](#) and [java.lang.Object](#) classes, respectively. Although it is technically possible in C++, subclassing C++ proxy classes for the purpose of making new non-proxy types is not fully discouraged and is not recommended. This is due to the fact that such derived C++ classes are limited in following ways: (1) Such types are not visible or directly instantiable from VM-based classes or environments. (2) Members fields of such types are not visible or modifiable from VM-based classes or environments. (3) Derived methods of such types are not called from VM-based classes or environments. (4) Implemented interfaces of such types are not recognized by VM-based classes or environments.

## Class Declaration

Classes are declared in the following form:

```
class classDeclarationModifiers className
{
    ...
};
```

where *classDeclarationModifiers* are one or more optional class declaration modifiers (see the subheading *Simulated Keywords: Class Declaration Modifiers*), and *className* is the name of the class being declared.

The class' fields and methods are declared within the class declaration block.

Instance (non-static) fields are declared in the following form:

```
accessModifier. typeName fieldName(methodArguments);
```

Static (class) fields are declared in the following form:

```
accessModifier. static typeName fieldName;
```

*accessModifier* is an optional field access modifier (see the subheading *Simulated Keywords: Method and/or Field Modifiers*), *typeName* is the field type, and *fieldName* is the name of the field. For example,

```
pi_packageprivate: jint myField;
```

Instance (non-static) methods are declared in the following form:

```
accessModifier. returnType methodName(methodArguments);
```

Static (class) methods are declared in the following form:

```
accessModifier. static returnType methodName(methodArguments);
```

*accessModifier* is an optional field access modifier (see the subheading *Simulated Keywords: Method and/or Field Modifiers*), *returnType* is the type which the method returns, *methodName* is the name of the method, and *methodArguments* are optional and are made up of one or more method arguments. For example,

```
pi_packageprivate: jint myMethod(jdouble value1, pi_double v2);
```

## **Class Declaration: Copy Constructor and Assignment Operator**

Many classes in C++ declare and define the copy constructor and the assignment operator. However, neither of these methods have exact equivalents in the Java language. Let's see why this is true.

The classic copy constructor in C++ is declared as:

```
ClassName(const ClassName& rhs);
```

where *ClassName* is the name of a class.

You can do something similar in Java:

```
ClassName(ClassName rhs);
```

But there are several important differences between the two versions though they might not be readily apparent. The C++ version passes a reference to a valid (non-null) object, and the Java version actually passes a reference, which may or may not refer to a valid (non-null) object. (Technically, it is possible to pass a null reference in C++, but this practice is almost universally discouraged. Keep in mind that references in C++ imply that they refer to valid (non-null) objects whereas pointers do not. See *Effective C++*, Second Edition, by Scott Meyers; Addison Wesley, 1998; p. 101)

The second reason has to do with managing the lifetime of the object. C++ by nature never does this, but Java by nature must always do this. The reference passed by the C++ version is just a thinly veiled pointer, actually a pointer by a different syntax. Even if the reference passed is to a managed object (i.e., a NewJNI reference type), there is no guarantee (in a multi-threaded application) that such reference will not be deleted during the execution of the constructor. “How is this possible?” you might wonder. It can and does happen when the object being passed to the constructor is a static class field and another thread reassigns it and causes it to be deleted while the constructor is still referring to it. Needless to say the result is undefined behavior, probably crashing your application.

The third reason has to do with `const`. In the classic C++ copy constructor the parameter being copied cannot be modified, but in Java it always can be because Java presently has no notion of `const`. (Of course, you use the C++ mutable keyword to get around this too but that’s another discussion entirely.) “But I can make the copy constructor take a non-`const` reference instead?” you might say. Such a copy constructor is declared as:

```
ClassName(ClassName& rhs);
```

where *ClassName* is the name of a class.

This has two problems. Most glaring, you can no longer copy `const` objects. That’s a big limitation. Don’t just skirt this limitation by casting away `const`. Doing so breaks the contract between your method and its caller, and attempting to modify unmodifiable objects results in dreaded undefined behavior. (See *Effective C++*, Second Edition, by Scott Meyers; Addison Wesley, 1998; Item 21, page 97.) The second problem is even if you never used the `const` keyword or any `const` objects you would still have to deal with the object lifetime management issue present above which means we’re back to where we started: neither the copy constructor or assignment operator have exact equivalents in the Java language.

But NewJNI in C++ has equivalents of the reference assignment operator and reference copy constructor. How can this opposite statement be true? Let’s delve into that.

The reference copy constructor in Java is declared as:

```
ClassName(ClassName rhs);
```

where *ClassName* is the name of a class.

The equivalent in C++ using NewJNI is:

```
ClassName(ClassNameR rhs);
```

Notice that the *ClassName* argument in the NewJNI version ends with a capital R. This is an important difference. The NewJNI version actually passes a reference to an object. Just like Java, it may or may not be null. Just like Java, the object's lifetime is managed. Just like Java, the object is modifiable. Case closed.

“But not so fast,” you might counter. “Every experienced C++ programmer knows that every class should declare its own copy constructor and assignment operator.” (See *Effective C++*, Second Edition, by Scott Meyers, Addison Wesley, 1998; “Item 11: Declare a copy constructor and an assignment operator for classes with dynamically allocated memory;” pp. 49-52.) And you would be correct. To make this easier and error-free, the following macro is provided. `PIE_HIDE_IMPLICIT_METHODS` is declared as:

```
PIE_HIDE_IMPLICIT_METHODS(ClassName);
```

where *ClassName* is the name of your class.

Now let's consider the assignment operator. How does the assignment operator member function in C++ differ from the reference assignment operator in Java? In C++ the assignment operator member function modifies the object being assigned (the left-hand side argument). In Java the reference assignment operator only affects the object reference not the object itself. Reassigning an object reference from one object to another in no way modifies either object. (Of course, information about objects, such as reference counts, may be affected.) This facet of the Java language makes it seemingly possible to modify immutable objects like `String`. For example,

```
String immutableString = “Strings are immutable!”;  
immutableString = “Strings are immutable?”;
```

Under closer examination no `String` objects are being modified—only the reference is being modified. Note that references in Java are always mutable unless they are declared `final`.

NewJNI provides this same functionality using reference types. The above example would be written in NewJNI as follows:

```
StringR immutableString = PI_T(“Strings are immutable!”);  
immutableString = PI_T(“Strings are immutable?”);
```

PI\_JT is a macro for making string literals into String objects. This is done by compiler sleight-of-hand in Java but must be done explicitly in C++.

The reference assignment operator is automatically provided for all reference types; nothing additional needs to be declared within the class itself. It is still necessary to declare and define reference types for all classes you define. For details on how to do this, see the subheading *Object References*.

Even though the C++ copy constructor and assignment operator member function do not have equivalents in Java, for design reasons or due to other constraints you may still need to implement these functions in their original C++ forms. This is perfectly legal. However, keep in mind that doing so may limit the convertibility of such logic back to Java, which may or may not be important to you.

Now what have we concluded? First, the C++ copy constructor and assignment operator member function do not have equivalents in Java. Second, the Java reference copy constructor and reference assignment operator have equivalents in C++ using NewJNI. Lastly, due to these differences it is best to always declare (but usually not define) the copy constructor and assignment operator and declare reference types for all the classes you define.

## Class Definition

The class definition must match the class declaration. The class' methods and static fields are declared outside the class declaration block.

Static fields are defined in the following form:

```
typeName className::fieldName = initializationValue;
```

*typeName* is the field type, *fieldName* is the name of the method, and *initializationValue* is an optional initialization value. For example,

```
jint MyClass::myField = 5908;
```

Methods are defined in the following form:

```
returnType className::methodName(methodArguments)  
{  
    ...  
}
```

*returnType* is the type which the method returns, *className* is the name of the class, *methodName* is the name of the method, and *methodArguments* are optional and are made up of one or more method arguments. For example,

```
jint MyClass::myMethod(jdouble value1, jdouble v2)
```

```
{  
    ...  
}
```

## Entry Point for Program Execution

In applications written for NewJNI the main entry point is `pi_main()`. It is declared as:

```
pi_int pi_main(PieMainContext context);
```

where *context* is an object that contains command line arguments and other information relating to application invocation. The `PieMainContext` type is a handle to a private internal object which is subject to change and should never be accessed directly.

The `pi_main()` function must be defined exactly once in each application developed for the NewJNI. However, it unnecessary to define `pi_main()` directly in your application. Instead NewJNI provides implementations of `pi_main()` for various startup options. Different startup options may be provided for each target operating system platform.

## Startup Options for Win32 Executables

For Win32 executable programs the startup options include *single-starter* and *multi-starter* for (GUI) event-driven applications and *shell single-starter* and *shell multi-starter* for shell-based non-GUI applications.

The *single-starter* and *multi-starter* startup options execute your GUI application outside the shell from which they are launched. This differs from most Java virtual machine implementations which execute GUI applications inside the shell from which they are launched. This is an important difference. The approach taken by *single-starter* and *multi-starter* causes applications to execute just like native Windows GUI applications, which immediately return control to the shell after being started instead of taking over the shell.

The *shell single-starter* and *shell multi-starter* startup option executes your application within the shell from which they are launched and do not return control to the shell until the application terminates. This is the same approach taken by most Java virtual machine implementations. This is also how native Windows shell commands and command line programs are executed.

**Note:** The present version supports only *shell single-starter* and *shell multi-starter* and does not support either *single-starter* or *multi-starter*. *Single-starter* and *multi-starter* may be supported in a forthcoming version.

## Startup Options for Win32 Executables: Single-Starter and Shell Single-Starter

*Single-starter* and *shell single-starter* requires that the starter class must be specified at compile-time. To specify the starter class define the symbol `PIE_STARTER_CLASS` to be the name of your starter class.

```
#define PIE_STARTER_CLASS StarterClassName
```

where *StarterClassName* is the name of your starter class. For example,

```
#define PIE_STARTER_CLASS MyApplication
```

If the starter class is declared in a namespace (besides the global namespace), the symbol `PIE_STARTER_CLASS_NAMESPACE` must also be defined to be the fully qualified namespace itself without the name of the class. If the class does not have a namespace (because it is in the global namespace), the symbol `PIE_STARTER_CLASS_NAMESPACE` should not be defined. For example,

```
// Example of class without namespace (actually in global namespace)
#define PIE_STARTER_CLASS MyApplication
// PIE_STARTER_CLASS_NAMESPACE should not be defined in this situation

// Example of class with namespace
#define PIE_STARTER_CLASS MyApplicationClassName
#define PIE_STARTER_CLASS_NAMESPACE com::nameofmycompany
```

The specified starter class must declare the traditional main method as follows:

```
public: static void main(::java::lang::StringAR arguments);
```

This main method is the effective entry point for your application. This is where execution begins in your application when it is started.

Lastly your application must include the single-starter logic. This must be done exactly once in each application. To satisfy this requirement it is recommended that you create a file named “PieMain.cpp” which adheres to the following form for *single-starter*:

```
#define PIE_STARTER_CLASS StarterClassName
#include "StarterClassName.h"
#include <pie/x-startup/awt/starter.cpp>
```

where *StarterClassName* is the name of your starter class. Note that both `PIE_STARTER_CLASS` must be defined and the header file containing the declaration of your starter class must occur before including the startup logic. Otherwise, the file will not compile successfully.

Here is the format of “PieMain.cpp” for *shell single-starter*:

```
#define PIE_STARTER_CLASS StarterClassName
#include "StarterClassName.h"
```

```
#include <pie/x-startup/shellstarter.cpp>
```

## Startup Options for Win32 Executables: Multi-starter and Shell Multi-starter

*Multi-starter* and *shell multi-starter* allows the starter class to be determined at run-time. The starter class must be specified at run-time instead of at compile-time like *single-starter*. This starter class is specified using the first command line argument. If the first command line argument does not specify a valid starter class the application will terminate abruptly.

Each potential starter class must declare the traditional main method as follows:

```
public: static void main(::java::lang::StringAR arguments);
```

This main method of the specified starter class is the effective entry point for your application. This is where execution begins in your application when it is started.

In addition, each potential starter class must be self-registering. A self-registering class declares that it is a main class along with its class declaration and implements (or defines) main class self-registration along with its class definition. To declare that a class is a main class include the following statement in the end of the class declaration:

```
NEWJNI_DECLARE_MAIN_CLASS();
```

For example,

```
class MyClass : public ::java::lang::Object
{
    ...
    NEWJNI_DECLARE_MAIN_CLASS();
};
```

To implement main class registration for a class use the `NEWJNI_DEFINE_MAIN_CLASS` macro as follows:

```
NEWJNI_DEFINE_MAIN_CLASS(className, classNameString);
```

where *className* is the name of the class, and *classNameString* is the string representation of the fully-namespace-qualified class name. For example,

```
NEWJNI_DEFINE_MAIN_CLASS(MyClass, "MyClass");
```

Lastly your application must include the appropriate startup logic. This must be done exactly once in application. To satisfy this requirement it is recommended that you create a file named "PieMain.cpp". Here is the format of "PieMain.cpp" for *multi-starter*:

```
#include <pie/x-startup/awt/multistarter.cpp>
```

Here is the format of “PieMain.cpp” for *shell multi-starter*:

```
#include <pie/x-startup/shellmultistarter.cpp>
```

## Initialization and Termination of NewJNI Runtime Library

Just like JNI, NewJNI must be initialized before it can be used. This is taken care of automatically by NewJ AppWizard-generated applications or by the starters detailed above. However, you may wish to use NewJNI in an existing application. If so, you will need to explicitly initialize NewJNI yourself. To initialize NewJNI, use the `initialize()` static method of `NewJNISystem`. It has two forms:

```
static bool initialize(  
    pi_int newjniMajorVersion,  
    pi_int newjniMinorVersion,  
    jint jniVersion,  
    const char* classPath,  
    jint* jniResult);  
  
static bool initialize(  
    pi_int newjniMajorVersion,  
    pi_int newjniMinorVersion,  
    JavaVMInitArgs& jvmInitArgs,  
    jint* jniResult)
```

Here is an example:

```
jint result;  
if (! newjni::NewJNISystem::initialize(0, 0, JNI_VERSION_1_2, “.”, &result))  
    {  
        // Failed to initialize NewJNI  
        exit(-1);  
    }
```

Note that the first two parameters for `initialize()` are reserved and should both be 0. The `initialize()` method returns true indicating that initialization has succeeded or false if it fails. You should always check for failure when calling `initialize()`.

Before your application completes, it is generally advisable to explicitly terminate NewJNI in order to free any resources it may have been using. To terminate NewJNI, use the `NewJNISystem::terminate()` static method as follows:

```
newjni::NewJNISystem::terminate();
```