



## Appendix A

# Interop 2: Bridging Java and Win32/COM

This chapter introduces Jawin, an open source architecture for Java/Win32/COM interop designed by the author.

## A.1 Overview

The Java platform standardizes the services and data formats that you need to assemble an application from separate components. The class loader architecture, type information, and reflective services, such as serialization, provide the infrastructure to load component code, configuration information, resources, and component data from disparate sources at runtime.

When you need to assemble applications across *different* component platforms, things fall apart. All platforms take their own approach to component services, and the differences from Java can be daunting. In the case of Microsoft's Win32 API and the Component Object Model (COM), the problems are not merely of academic interest. Win32 and COM are associated with the dominant Microsoft Windows family of operating systems. Because the Windows family is so prevalent, most organizations will need to deal with code written for these systems, even if they have made a strong commitment to Java.



One approach to calling Win32 and COM components is to use the Java Native Interface (JNI), as described in Chapter 6. JNI does provide the necessary tools, but it is tedious to use for any but the smallest projects. When you use raw JNI, you have to write a large amount of infrastructure code every time you want to cross the boundary between Java and Win32 or COM. Moreover, JNI is a generic architecture for calling native code, and therefore it does not include any features that deal with the specific problems of the Win32 platform. For these two reasons, organizations that need to implement substantial communication between Java and Win32/COM components will want to look for other answers.

This appendix will demonstrate an alternative to JNI by describing a higher-level strategy for in-process interoperation between Java, Win32, and COM components. This strategy uses translucent stubs (introduced in §A.2) that bridge the differences between component platforms. These stubs are important because they sit between components from different platforms and hide the details of cross-platform communication. §A.3, §A.4, and §A.5 present the key differences between Java, Win32, and COM as component platforms and describe how translucent stubs might resolve these problems.

Most of a marshalling layer implementation is generic and can be shared by all components. However, each component interface will need its own interface stub, which must either be developed manually or generated from type information. §A.7 discusses how to generate stubs for Win32 and COM interfaces.

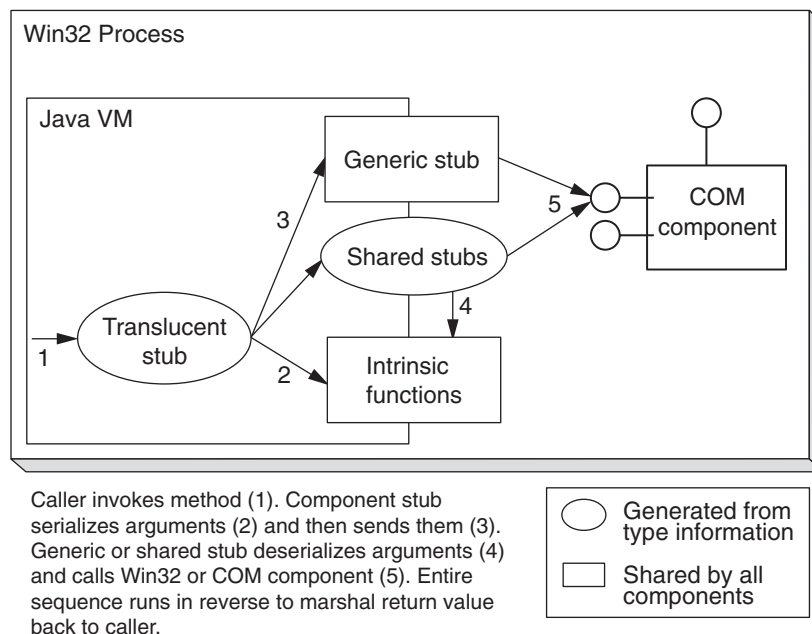
For concrete examples, this chapter uses Jawin, an open source Java-to-Win32/COM marshalling layer developed by the author. However, the emphasis here is on general concepts, not on the specifics of the Jawin implementation. For a list of other Java/Win32 interop solutions, see [JavaWin32].

## A.2 Translucent Stubs

A translucent stub is the visible part of a marshalling architecture. A marshalling architecture moves a method call from one environment to another by executing the following series of steps.

1. Convert a method invocation into a request message.
2. Deliver the request message to a target environment.
3. Convert the request message into a method stack in the target environment.
4. Invoke the method.
5. Convert the return value(s) or exception(s) into a response message.
6. Deliver the response message to the source environment.
7. Convert the response message back into the types expected by the caller.

Figure A-1 illustrates these steps.



**Figure A-1 Marshalling a call from Java to COM**

Stubs vary in how well they hide the details of the steps listed above. JNI stubs are very simple, leaving the programmer explicitly aware of all of the steps listed above. On the other hand, you can define a *transparent* stub as one that completely hides the details of communication. Transparent stubs might seem ideal, but they are usually difficult (or impossible) to implement. A *translucent*

stub fits somewhere between the extremes. It hides the details of communication that are easily hidden, but exposes the communication layer in some places.

JNI provides only a minimal interface for communicating between Java and native code. As Chapter 6 demonstrated, you must write hand-tuned code to deal with such basic issues as converting parameter types, manipulating arrays, and reporting errors. Worse, you must duplicate this code for every single native method. When you write JNI code, you are always acutely aware that you are working near the boundary of the Java platform, and because of this, you must be skilled in both Java and the native platforms.

If transparent stubs could be created easily, the problems of JNI would be neatly solved. Clients would not have to write any additional code to call Win32/COM components. More importantly, clients would not have to know any details about the Win32/COM platforms or even be aware that these platforms were in use.

How hard is it to create a transparent stub? There are three things to worry about when you are creating transparent stubs between Java, Win32, and COM:

1. Platform impedance. Each platform makes certain fundamental assumptions that are not valid elsewhere, thus forming an “impedance mismatch” between the platforms. Transparent stubs must hide these differences. For example, COM components indicate errors with numeric codes, which a transparent Java stub would hide by translating the error codes into Java exceptions.
2. Generating per-interface stubs. Each different “interface,” however that concept maps to a particular platform, implies a different stub. These stubs are very similar to RMI stubs. Therefore, an interop solution needs to include a code generator. This could be a developer tool similar to `rmic`, or a runtime API akin to dynamic proxies.
3. Performance. The stubs have to meet performance criteria, which are specific to an application.

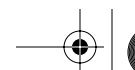
To determine whether transparent stubs are worth the effort, compare the costs of building the stubs with the benefits they provide. Is it easier to efficiently generate stubs that cope with various platform impedance issues, or is it easier to just brute-force your way with raw JNI?

Translucent stubs are based on the observation that some platform impedance problems are more difficult to solve than others. The stubs are called translucent because they hide *most* of the details of calling into native code. When an idiom has an obvious mapping from one platform to another, such as the error codes and exceptions mentioned previously, then it should be hidden in the stub layer. When an idiom does not translate well to another platform or you must understand it to use the component, it is more appropriate to expose the idiom directly.

Java RMI provides a good example of translucent stubs. RMI stubs hide the details of calling an object in another virtual machine, usually over a network. Using RMI is much easier than manually converting method invocation into socket communication. However, RMI stubs are not completely transparent. Object parameters passed to RMI methods must be serializable so that they can be transmitted over the network. RMI stubs will enforce this rule by throwing an exception, which breaks transparency and reveals to the client that RMI is involved.

In another case, the designers of RMI deliberately broke with transparency. They made it so all remote methods must be declared to throw the checked exception `java.rmi.RemoteException`. This reflects the underlying reality that even if the method succeeds, the stub communication may fail. Because the exception is checked, all clients must deal with it, which makes the presence of RMI obvious. It would have been just as easy to make `RemoteException` unchecked, which would have been more transparent. However, this would only have lulled developers into a false sense of security. The possibility of communications failure is so important that it outweighs the convenience of simpler, more transparent clients.

§A.3 describes the sources of platform impedance between Java, Win32, and COM. In this discussion, you will see how one particular stub architecture [Jawin] deals with these issues. The particular choices made in this implementation are reasonable, but not inevitable. Different development teams will assign different weights to the issues.



## A.3 Platform Impedance

This section describes the facets of the COM and Win32 platforms that make interoperation with Java difficult. Table A–1 summarizes the chief sources of platform impedance. The COM approach is explored in more detail in §A.4, and the Win32 approach is discussed in §A.5. This is obviously not a complete introduction to these complex technologies. For more on COM see [Box98]; for more on Win32 see [Ric99].

**Table A–1 Platform Impedance: Java, Win32, and COM**

Concern	Java Approach	Win32 Approach	COM Approach
Loader architecture	Class loaders	Explicit paths	Registry lookup
Metadata/ Type info	Extensive	Minimal	Partial
Object lifecycle management	Implicit, GC	Explicit and ad hoc	Explicit and reference-counted
Type discovery	Per-class	Ad hoc	Per-instance
Error reporting	Throwable	GetLastError et al.	HRESULT et al.
Thread affinity	Explicit	Explicit	Apartments
Security model	Code source and signer	User principal	User principal

The examples in §A.4 and §A.5 show client-side syntax for dealing with these problems from Java, and subsequent sections show one approach to implementing this client-side syntax.

## A.4 The Component Object Model

The Component Object Model (COM) defines a binary representation for interface contracts between components. Above that core level, COM includes a runtime with many services that are useful in assembling applications from



components. Where the emphasis in Java is one language for many platforms, COM aims at many languages running on one platform.

Different languages use the COM runtime in different ways, which makes learning COM arduous. If you look past the trappings of particular languages, the COM specs and API docs define a set of core services analogous to many of those described in this book:

- COM defines a local loader based on lookup information stored in the Windows Registry.
- COM defines two different type information formats that are not entirely compatible.
- COM defines a reflection-like API for accessing type information.

While these services fill similar roles to their Java counterparts, they are different enough to introduce quite a bit of platform impedance.

#### A.4.1 The COM Loader

The COM loader loads COM objects by their unique name, which is a 128-bit identifier called a GUID (Globally Unique ID). The loader uses this name to locate the appropriate binary by consulting the Windows Registry. The Registry typically also contains a human-friendly name called a ProgID.

GUIDs and ProgIDs do not have an obvious analog in Java. The GUIDs do not map well to class names, and the ProgIDs may not be unique. Most importantly, any name mapping would replace a name that has some meaning in the COM world with a new name that has meaning only as defined by the stub architecture. The most appropriate thing to do is to expose these constructs directly to Java clients as shown here:

```
//create an instance of MS-Word from Java, using the ProgID:
WordApp wd =
    (WordApp) Ole32.CoGetObject("new:Word.Application", ...);

//using the GUID:
GUID wordGUID = new GUID
    ("{"000209FF-0000-0000-C000-000000000046}");
WordApp wd =
    (WordApp) Ole32.CoCreateInstance(wordGuid, ...);
```

`WordApp` is a generated stub for the top-level COM interface to Microsoft Word. The `Ole32` class is a hand-coded API stub that is part of Jawin.

#### A.4.2 COM Type Information

Type information poses a trickier problem. COM type information comes in two important formats, the Interface Definition Language (IDL), and the type library. C and C++ developers use IDL to describe interfaces, as shown in Listing A–1.

##### Listing A–1 A Simple IDL Fragment

```
HRESULT  
CrunchArray([in] int sizeIn, [in, size_is(size)] int* in,  
            [out] int* sizeOut, [out, size_is(1,sizeOut)] int** out);
```

IDL method declarations include method names, return types, and argument types. A tool that could read the IDL text file could use this information to generate Java stubs.

There are two problems with using IDL files to generate stubs. First, many COM implementation languages do not use IDL files at all; instead, they describe interfaces with a *type library*. Type libraries are not as expressive as IDL and support only the most commonly used types. Type libraries can be stored standalone in type library (TLB) files or bound into the application binary as resources. The `ITypelib` and `TypeInfo` interfaces, COM's version of reflection, can be used to extract the information from a type library. This turns out to be a much easier approach to stub generation than parsing IDL. You do not need to write a parser, and type libraries are much more widely available. Even if there is only an IDL file available, you can use the `midl` compiler to generate a type library.

The second problem with IDL is that the vocabulary is too flexible. Because it can describe interfaces for pointer-based languages such as C and C++, IDL includes a number of constructs that do not have any trivial mapping to Java—or to most COM languages, for that matter. For example, the `CrunchArray` method in Listing A–1 uses two `out` parameters to “return” multiple values to the caller. A straight translation to Java will not work since Java does not support `out` parameters. In this example, the right thing to return would be an



array, but IDL can express even more complex signatures that have no obvious Java mapping.

Most integration tools deal with the complexity of IDL by avoiding it. They support only the subset of types that can be expressed in a type library. Many tools handle an even smaller subset of types, those that are `VARIANT`-compatible. The `VARIANT` data type is a union of types that are available to scripting languages. Although `VARIANT`-compatible types are a small subset of full IDL, this limitation is not as bad as it seems. A large fraction of all COM interfaces voluntarily limit themselves to these types anyway so that they can be used from scripting languages. The Jawin architecture is capable of supporting all IDL types, but the implementation is complete only for the most commonly used types.

#### A.4.3 COM Object Lifecycle

COM uses reference counting to manage object lifecycle. All COM interfaces extend the base interface `IUnknown`, which has two reference counting methods:

```
ULONG AddRef();  
ULONG Release();
```

When clients acquire an interface pointer, they must call `AddRef`. When they are finished with the interface, they must call `Release`. COM objects manage their own lifetime based on these hints. The most common approach is for the object to maintain a reference count and delete itself when the count reaches zero, although more esoteric approaches are possible. The client does not care what the object actually does in response to these methods; it simply must follow the rules in calling them. There are three ways that a Java client might deal with reference counting COM components:

1. Java stubs could implement the `finalize` method, which would call `Release` when triggered by the garbage collector.
2. Java stubs could provide an explicit `Release` method that clients must remember to call.
3. Stubs could provide both `finalize` and an explicit `Release`.

Option 1 is the most transparent. Java programmers are not used to reference counting, so why make them remember this detail? Unfortunately, this does not have acceptable performance. Garbage collection is not deterministic, so there is no guarantee when `finalize` will run, if ever. COM objects would be held in memory for an indeterminate amount of time, even after they were no longer needed. Worse yet, `finalize` might be called on the wrong thread.

Option 2 requires that clients remember to call `Release` when they are finished using a Java stub. This is the most efficient approach. However, if the client forgets to call `Release`, the object will be leaked. With option 3, `finalize` serves as a backup in case clients forget. This two-pronged approach is standard throughout the core Java API where native resources are used. In the core API, Java classes that manipulate sockets, files, database connections, or other native resources provide a well-known method, typically named `close`, which clients call to free the native resource. Jarwin provides a `close` method, but does not use `finalize` because of the threading issue.

#### A.4.4 COM Type Discovery

In COM, type is a property of a particular instance, not of an entire class. A particular instance can implement one or more interfaces. Interfaces are uniquely named by 128-bit GUIDs called IIDs. To discover whether an object supports a particular interface, you call the method

```
HRESULT QueryInterface(REFIID iid,  
                        [out, iid_is(riid)] void *ppvResult)
```

Based on their responses to `QueryInterface`, different instances of the same class may implement different interfaces. This is the opposite of Java, in which the list of implemented interfaces is a fixed property of the binary class format.

Java's cast operator relies on the fixed nature of the binary class format, and therefore cannot work with transparent COM stubs. There are two options: Hack a virtual machine to change the semantics of casting, or simulate casting with a

method call. The former option violates the Java license,<sup>1</sup> so there is no hope for modifying the virtual machine. Casting from one interface to another requires a method call:

```
SomeComInterface itf1 = getSomeComInterface();

//this won't work:
//OtherComInterface itf2 = (itf2) itf1;

//you must explicitly call COM's QueryInterface
OtherComInterface itf2 = (itf2)
    itf1.QueryInterface(itf2.class);
```

Notice that this notion of “casting” also muddies the concept of identity. The variables `itf1` and `itf2` refer to the same COM object, but they are different stubs. The expression `(itf1 == itf2)` will evaluate to `false`.

#### A.4.5 COM Error Handling

COM provides several levels of support for error reporting. All COM interface methods return an unsigned 32-bit type called an `HRESULT` to indicate success or failure. The significant `HRESULT` values have an associated text message that can be retrieved via the Win32 API call `FormatMessage`. Applications that want to provide more specific information about an error can populate an “error object” which can be placed in thread local storage by calling `SetErrorInfo`. Clients can then “catch” the error object by calling `GetErrorInfo`.

All of COM’s error information can be mapped into Java by simply creating a subclass of `Exception` that has data members for the `HRESULT`, text message, and any data from the error object. As a result, Java programmers can handle exceptions from Java/COM stubs just like they would for any other Java class. The `COMException` class in Listing A-2 is a simple wrapper for COM error information. The interesting design decision is whether to make the `COMException` class a checked or unchecked exception. In Jawin, `COMException` is a

---

1. This was one of the elements of Sun’s lawsuit over the Microsoft VM, which modified casting to transparently support COM semantics.

checked exception. This design choice is based on the similar decision in the design of `RemoteException` for RMI. Even if the method itself succeeds, there may be communication errors in the stub layer.

#### Listing A-2 COMException

```
package com.develop.jawin;

public class COMException extends Exception {
    public final int hresult;
    public static final int E_UNEXPECTED = 0x8000ffff;
    public COMException() {
        this(E_UNEXPECTED);
    }
    public COMException(int hresult) {
        //code to get error string not shown
        this.hresult = hresult;
    }
    public COMException(int hresult, String text) {
        super(text);
        this.hresult = hresult;
    }
    public COMException(String text) {
        this(E_UNEXPECTED, text);
    }
    public String getMessage() {
        return Integer.toHexString(hresult) + ": "
            + super.getMessage();
    }
    public COMException(Throwable t) {
        this(E_UNEXPECTED, t.getMessage());
    }
}
```

#### A.4.6 COM Thread Affinity

From a Java programmer's perspective, the most unusual feature of COM is its built-in support for thread-affine components. Java, Win32, and COM all define some resources as thread-affine—those resources that can be used only from a subset of the threads in the process. There are two historical reasons for this:

1. User-interface subsystems are often built to run on a single thread to simplify the programming model for UI developers.
2. Some components do not use the concurrency protection mechanisms of the underlying platform, and therefore they are unsafe when called from more than one thread at a time. Most C++ and Java programs suffer from this problem.

In Java and in Win32, there is no special support for thread-affine components. You simply have to be careful. If you break the rules and call a component from the wrong thread, the resulting behavior is typically undefined. You might get lucky and crash your application, or you may get unlucky and have data corruption that goes unnoticed.

COM provides apartments to deal with thread affinity. An *apartment* is a logical subspace of a process within which all objects expect the same thread semantics. Method calls *across* apartments go through stubs<sup>2</sup> (called proxies in the COM world) that do any extra work necessary to guarantee the correct thread semantics, such as switching calls onto an appropriate thread. Objects can live in a single-threaded apartment (STA) if they want to be called from only one thread throughout their lifetime; or, they can live in a multi-threaded apartment (MTA) if they want to be called from multiple threads and might execute blocking calls. If an object knows that it will never need to make a blocking call, it can safely “visit” either an STA or the MTA and have no thread affinity. COM objects with no thread affinity are called agile, or are said to “aggregate the free-threaded marshaller.”<sup>3</sup>

With COM+ 1.0, the apartment story is even more complex. The apartment model is extended to a more general *context model*. Objects can request that they live in a particular context, and inbound method calls will pass through a proxy that sets up the required context. This is very similar to EJB, where

---

2. These stubs are very similar to the stubs that are the subject of this chapter. They are generated from type information and exist to bridge between incompatible components—in this case *within* a single component platform.

3. This technology describes how an object implementor defeats thread affinity. Aggregation is a reuse mechanism in COM, and the free-threaded marshaller is a component that prevents the creation of cross-apartment stubs. This causes an object to belong to all apartments (or none) depending on your perspective.

method calls pass through container-generated code that sets up the transaction and security context. The addition of contexts necessitates yet another apartment type. The thread-neutral apartment (TNA) is home to objects that do not have thread affinity but need to have stubs to initialize their context.

Apartments are quite complex, and many trees have been felled describing them. There are three possible approaches to take when dealing with this complexity from Java:

1. Require that Java clients understand COM apartments and use them correctly.
2. Require that Java clients treat all COM components as thread-local.
3. Build logic into the stub layer that analyzes the current thread on each method call, and uses an appropriate COM proxy.

None of these options is perfect. Option 1 is a non-starter; it is completely unrealistic to expect that Java programmers learn the details of COM apartments just to use a COM object. Option 2 provides a very simple rule that Java clients can deal with, but it is overly restrictive in many cases. Option 3 is entirely transparent to Java clients, but the performance hit is significant in some cases.

Since none of the three options is perfect, Jawin supports more than one. Option 3 is the default: Stubs always guarantee that COM threading rules are followed. This will work correctly in all cases but may be slow. If you know that you plan to make repeated calls from the same thread, Jawin provides a helper method called `contextLocalize`. Calling `contextLocalize` turns off a stub's built-in apartment support. Calls through the stub will execute more quickly, but they are only guaranteed to work from the current thread.

#### **A.4.7 COM Security**

The security architecture is the most troubling source of impedance mismatch between COM and Java. COM security is based on the user identity currently associated with a thread. Java assigns permissions based on the code source, that is, the location (URL) that code came from and the signers of the code (if any). In theory, there is no problem with securing COM objects using Java permissions. It

is simply a matter of defining some appropriate `Permission` subclasses, and calling `checkPermission` inside the generated stubs. But, you would have to write this code by hand. There is no straightforward way to generate security-aware stubs because there is no security metadata. Because there is not a good solution, most interop products (including Jawin) either ignore security or build special cases by hand.

## A.5 Win32 Dynamic Link Libraries

Dynamic Link Libraries (DLLs) are the basic mechanism for component reuse in the Win32 family of operating systems. DLLs have been around much longer than Java or COM, so when I describe the DLL architecture as a component platform, I am fitting new terms to an old technology. While the goals of component programming are less fully realized in the DLL architecture, the key elements are still visible.

### A.5.1 The DLL Loader

The loader architecture for DLLs is quite simple. First, call `LoadLibrary` to locate a binary by its name and location in the file system. Then, call `GetProcAddress` to locate a particular function entry point in the library, either by name or by ordinal (numeric address). Cast the result of `GetProcAddress` to an appropriate function pointer, and off you go, as shown in Listing A-3.

#### Listing A-3 Dynamic Loading of a DLL Entry Point

```
//dynamically loading and calling MessageBoxW:
typedef WINUSERAPI int WINAPI
    MBFUNC(HWND, LPCWSTR, LPCWSTR, UINT);
HMODULE hm = LoadLibrary("USER32.DLL");
MBFUNC* MsgBox = (MBFUNC*)GetProcAddress(hm, "MessageBoxW");
MsgBox(0,L"Hello",L"Dynamically Loaded", 0);
```

The important difference from Java is that DLLs deal in functions, not objects. Since Java does not allow freestanding functions, the obvious mapping is to represent these functions as static methods on some Java class. This is Jawin's approach.

How should DLL entry point functions be grouped into Java classes? One approach would be to have a single Java class with all the stub functions for a particular DLL. For example, `User32.java` would contain stubs for all the functions from `User32.dll`. This approach leads to very large stub classes—`User32.dll` contains around 700 function entry points. An alternate approach is to group smaller sets of related functions under some meaningful name. Jawin usually takes the latter approach; for example, all the Registry functions are grouped in `Registry.java`.

### A.5.2 DLL Type Information

A compiled DLL does not expose any useful type information other than the names of methods. In this respect DLLs are very primitive compared to either COM or Java. If you look back at Listing A–3, you will see that loading the `MessageBoxW` function is not type-safe. All DLL entry points look the same until you cast them. If you cast wrong, chaos ensues.

Java clients expect and deserve more type safety than this. In order to generate strongly typed stubs, you must obtain type information from somewhere. There are several possibilities:

- IDL files
- Type libraries (TLBs)
- Header (`.h`) files
- Custom type formats

IDL files and type libraries you remember from the discussion of COM type information. Both IDL and type libraries can describe DLL entry points. However, few DLLs actually ship with this information. Indeed, most DLLs are described by header files. The C/C++ header files ship with the Windows SDK, but they are more difficult to parse.

Even if you have successfully obtained and parsed an IDL, TLB, or header file, you may have trouble generating a Java-friendly signature for some Win32 functions. Consider the Win32 API `GetTokenInformation` shown in Listing A–4.



### Listing A-4 Win32's GetTokenInformation

```

BOOL GetTokenInformation(
    HANDLE TokenHandle,           // handle to access token
    TOKEN_INFORMATION_CLASS TokenInformationClass, // token type
    LPVOID TokenInformation,      // buffer
    DWORD TokenInformationLength, // size of buffer
    PDWORD ReturnLength           // required buffer size
);

```

Logically, this function returns an object whose type is determined by the `TokenInformationClass` flag. However, this “object” takes the form of an opaque array of bytes copied into the `TokenInformation` argument. A Java client would expect to see not these raw bytes, but instead an instance of a Java class. Because the information needed to make this conversion is not part of the method signature, a simple type library would be inadequate. You would need to customize the type information to describe this conversion, which the method signature does not capture.<sup>4</sup> Given these issues, it might be easier to enter the type information by hand, using a custom format.

Most interop products fail to support any of these options and are unable to generate stubs for Win32 DLLs. The current version of Jawin requires that you write Win32 stubs by hand, although a future version may utilize a custom XML-based type format.

### A.5.3 DLL Object Lifecycle

Many DLL entry points define some notion of an “object” whose lifecycle needs to be managed. For example, the Registry function `RegOpenKey` returns an opaque handle that represents a key in the Registry. When you are finished with the key, you should call `RegCloseKey` to release the resource. Java stubs could hide this detail from clients by implementing `finalize` to release the resource, but this is a bad idea. As mentioned earlier in the discussion of COM lifecycle, `finalize` is unreliable. Even though it makes the programming model more

---

4. Neither COM nor Java metadata captures this sort of information either, so aren't all the component technologies equally vulnerable to this problem? In theory, yes; but in practice, no. The coding style (modulo syntax) used in `GetTokenInformation` is legal in Win32, Java, or COM, but though it is very common in the Win32 API, it is frowned upon in both the Java and COM worlds.

difficult, clients should be forced to call the lifecycle management APIs for such objects. These calls can be exposed directly, which is how Jawin handles the problem. If the stub layer manufactures a Java object to wrap the handle, then the lifecycle management API call should be hidden behind a `close` method. In the latter case, `finalize` can be used as a failsafe.

#### A.5.4 DLL Type Discovery

Because DLL entry points are not objects, they do not implement interfaces or support any notion of inheritance.<sup>5</sup>

#### A.5.5 DLL Error Reporting

DLLs do not enforce a standard scheme for reporting exceptional conditions. In the Win32 SDK, API calls report errors in several different ways:

- Some functions return zero on failure. A numeric code with more information is available by calling `GetLastError`. After calling `GetLastError`, you can obtain a string describing the error with `FormatMessage`. `GetTokenInformation` is one example of this approach.
- Some functions return zero on success. Again, more information is available through `GetLastError`. `RegCloseKey` is one example.
- Some functions return an error code such as an `HRESULT` directly. Most of the support API for COM fits in this category.

To add to the confusion, third-party DLLs are free to invent more unusual schemes. It should be straightforward to convert any documented exception reporting scheme into an exception for Java clients. Jawin handles the three standard types shown above, and it is extensible to deal with others.

#### A.5.6 DLL Thread Affinity

DLLs do not provide any special support for thread affinity, and neither does Java, so there is no impedance mismatch here.

---

5. DLL entry points into C++ libraries may provide access to C++ objects that do implement multiple interfaces. Jawin does not directly support non-COM C++ objects because there is relatively little demand for this ability. It would be straightforward (but tedious) to extend Jawin's COM support to also provide direct C++ support.



### A.5.7 DLL Security

The issues with the DLL security model are the same as with COM; see §A.4.7.

## A.6 Marshalling Architecture

A marshalling architecture ships method calls from one environment to another and provides some degree of help with the platform impedance issues discussed earlier. As mentioned in §A.2, marshalling involves seven steps:

1. Convert a method invocation into a request message.
2. Deliver the request message to a target environment.
3. Convert the request message into a method stack in the target environment.
4. Invoke the method.
5. Convert the return value(s) or exception(s) into a response message.
6. Deliver the response message to the source environment.
7. Convert the response message back into the types expected by the caller.

In order for this communication to occur inside a single process space, the marshalling layer must be built on top of JNI. There are two distinct approaches to marshalling a message from Java into native code. The shared stub approach, discussed in §A.6.1, lets JNI do the marshalling, but it requires that each unique method signature be manually coded in JNI. A generic stub, described in §A.6.2, marshals method calls and results in an opaque array of bytes. All method calls can then be implemented by a single native method.

### A.6.1 Shared Stubs

The idea behind shared stubs is simple. Although the theoretical number of different API signatures is practically infinite, most APIs actually use one of a very small number of signatures. For example, you have probably coded a method that takes an `int` and returns an `int`. However, have you ever written a method that takes `int`, `float`, `int`, `int`, `int` and returns `double`? Probably not.

If the total number of different API *signatures* is small relative to the total number of different API *methods*, then it makes sense to develop and test a



small set of native entry points, one for each signature. Once these entry points are in place, then you can add new methods without having to develop or test any new native code.

For example, consider the Win32 APIs listed in Listing A–5. These functions, selected from several different areas of the Win32 API, all appear to have different signatures. However, the `HGDIOBJ`, `int`, `LPVOID`, `HRESULT`, `BOOL`, `HWND`, `HANDLE`, `LONG`, and `HKEY` types all could be represented by the Java type `int`. As a result, these APIs could all share a single native stub method.

#### Listing A–5 Many APIs Can Share Stubs.

```
HGDIOBJ GetStockObject(int fnObject);
HRESULT CoInitialize(LPVOID reserved);
BOOL UpdateWindow(HWND hwnd);
BOOL DeregisterEventSource(HANDLE hEventLog);
LONG RegCloseKey(HKEY hKey);
```

Listing A–6 shows two shared stubs: one that handles one `int` argument, and a similar method that handles a single `String` argument. These stubs use a naming convention of `invoke{X*}_X`, where each `X` is replaced by a letter indicating the argument type, in this case `I` for `int` and `G` for `String`. Each stub takes zero or more parameters for the method arguments, plus the two special arguments `flags` and `func`. The `flags` control interpretation of the return value, and they correspond to the different error handling schemes for DLLs listed in §A.5.5. The `func` argument is the address of the API function.

#### Listing A–6 Shared Stub for Win32 APIs with a Single Integer Parameter

```
//Client code example: calling CoInitialize
Ole32.CoInitialize();

//Implementing the interface stub for CoInitialize
public class Ole32 {
    public static void CoInitialize() throws COMException {
        FuncPtr fp = new FuncPtr("OLE32.DLL", "CoInitialize");
        SharedStubs.invokeI_I(0, fp.getPeer(), CHECK_HR);
    }
    //etc.
}
```

```
//implementing the interface stub for the Registry APIs
public class Registry
{
    static private final FuncPtr fpCK;
    static {
        fpCK = new FuncPtr("ADVAPI32.DLL", "RegCloseKey");
    }
    public static void CloseKey(int key)
        throws IOException, COMException
    {
        SharedStubs.invokeI_I(key, fpCK.getPeer(), CHECK_W32);
    }
    //etc.
}

//excerpt from com.develop.com.marshall.SharedStubs;
public class SharedStubs {
    public static native int invokeG_I(int arg0, String arg1,
                                       int func, int flags);
    public static native int invokeI_I(int arg0,
                                       int func, int flags);
    //etc.
}
```

The native implementations of the shared stubs, shown in Listing A-7, are tedious but straightforward. Each stub executes the same basic series of steps:

1. Convert arguments to Java types if necessary.
2. Cast the `peer` to a function pointer and invoke the function.
3. Do any special processing of the return value.
4. Free any resources allocated in step 1.

#### **Listing A-7 Implementation of Shared Stubs' Native Methods**

```
typedef HRESULT (__stdcall * FTYPE1)(int);

inline bool checkRet(int ret, int flags) {
    switch (flags) {
        case 0:
            return true;
        case 1:
            if (!ret) {
                JNCOMException::SetLastError();
            }
    }
}
```

```

        return false;
    }
    return true;
    case 2:
    if (FAILED(ret)) {
        JNIComException::SetContextException(ret);
        return false;
    }
    return true;
    case 3:
    if (ret != ERROR_SUCCESS) {
        JNIComException::SetContextException(ret);
        return false;
    }
    return true;
    default:
    JNIComException::SetContextException(
        "Invalid code in checkRet");
    return false;
}
}
JNICALL jlong JNICALL
Java_com_develop_com_marshall_SharedStubs_invokeI_1I
(JNIEnv * pEnv, jclass, jint arg0, jint peer, jint flags)
{
    int ret = ((FTYPE1)peer)(arg0);
    checkRet(ret, flags);
    return ret;
}
JNICALL jlong JNICALL
Java_com_develop_com_marshall_SharedStubs_invokeG_1I
(JNIEnv * pEnv, jclass, jstring arg0, jint peer, jint flags)
{
    CComBSTR bs0;
    bs0.Attach(pEnv->jstobs(arg0));
    int ret = ((FTYPE1)peer)((int)bs0.m_str);
    checkRet(ret, flags);
    return ret;
}

```

For a stub that handles only primitive numeric types, steps 1 and 4 disappear and the implementation is entirely trivial. For slightly more complex types,

such as strings, the marshalling layer uses a set of helper APIs called *intrinsic functions* to perform data conversions.

An intrinsic function is a hand-coded function that implements some atomic action inside the marshalling layer, such as streaming a particular data type or converting a numeric error code into a Java exception. The intrinsic functions get their name because they are built into the marshalling layer. More complex marshalling tasks, such as streaming a large `struct`, do not require hand-coded functions. They take the form of several intrinsic function invocations in sequence and are generated from type information.

In Listing A-7, the `invokeG_I` implementation uses a helper class `CComBSTR` as an intrinsic function. In this case, the “function” is actually a class because `CComBSTR` is already available as part of the Active Template Library (ATL), which Jawin uses. The `CComBSTR` class automates converting a Java string into a `BSTR`, which is the most common string format in COM programming.

When does a particular marshalling task deserve its own intrinsic function, and when should it be composed from calls to lower-level intrinsic functions? The boundary is arbitrary. High-frequency data types, such as strings and arrays, deserve custom functions both for convenience and for performance reasons. However, there is a strong motivation to minimize the number of intrinsic functions, since some human must write each one. Most intrinsic functions come in pairs: a Java function to convert from Java to Win32/COM, and a native function to go in the other direction, from Win32/COM to Java. Never having to hand-code a native function is a key goal of building a marshalling layer in the first place. Wherever possible, Jawin executes more complex marshalling tasks by calling preexisting intrinsic functions.

For all their value, shared stubs are unlikely to handle every possible API. Methods with a large number of arguments, or with more complex structured types, require more complex stubs—and are therefore likely to not have shared stubs at all. Of course, some stubs are better than no stubs. If nine of ten needed APIs can be accessed through a preexisting shared stub, then you have achieved a 90 percent reduction in the native code that you have to write.



### A.6.2 A Generic Stub

Shared stubs implement one specific signature per stub. An alternate approach is to use a generic stub that can marshal any method call. A generic stub converts a method call into a serialized request message, which is then transmitted to the destination object. In addition to the request, the generic stub must also transmit some instructions that describe the method signature to be called. Using the request and the instructions, the generic stub builds the native call stack and invokes the method. Return values and exceptions are then serialized into a response, and returned to the client.

In theory, a generic stub could encode everything about the method call into a single array, and be declared like this:

```
public native byte[] genericInvoke(byte[] request);
```

In practice, the method call is likely to be encoded into more than one argument, for the convenience of the developer. Jawin's generic stub has this signature:

```
package com.develop.com.marshall;

public class GenericStub {
    public static byte[] win32Invoke(int peer, String inst,
                                     int stackBytes, int totalBytes,
                                     byte[] request, Object[] ObjectArgs);
    //remainder omitted for clarity
}
```

The `request` array is the serialized call, `totalBytes` is the number of relevant bytes in `request`, and the `peer` is the native function pointer to invoke. The separate array for Java object arguments is necessitated by JNI. JNI requires that objects passed to native code must be passed as object references, so the objects travel in their own separate array `ObjectArgs`. The object array also does double duty for any Java objects to be returned to the caller. The `StackBytes` and `inst` arguments are described in the next section.





### A.6.3 Instruction Strings

The other two parameters, `stackBytes` and `inst`, tell the marshalling layer how to build the native call stack. The `stackBytes` value is the size of the call stack. The implementation of `win32Invoke` will allocate a call buffer of this size on the stack. Then, the `request` array must be unmarshalled into the buffer. The `inst` argument is an *instruction string* that tells the marshaller how to copy the `request` array into the call buffer.

An instruction string is a sequence of characters that drives a state machine inside the marshaller. The state machine processes the instruction string to create a call stack. If all the arguments to a method are primitive types, then the instructions are trivial: Simply copy the request directly into the call buffer. If the arguments to a method are COM interface pointers or data structures, then the instructions get more complex. For example, consider the Jawn code to call the `MessageBoxW` function that appears in Listing A-8.

#### Listing A-8 Marshalling MessageBoxW

```
//the MessageBoxW signature in C:
WINUSERAPI int
WINAPI MessageBoxW(HWND, LPCWSTR, LPCWSTR, UINT);

//Client code: calling MessageBoxW from Java
User32.MessageBoxW("Hello World", "Jawn");

//Implementing the interface stub. This code might be
//hand-written or generated from type information.
public class User32 {
    static final int mstackMessageBoxW = 16;
    public static void MessageBoxW(String msg, String title)
        throws COMException
    {
        FuncPtr fp = new FuncPtr("USER32.DLL", "MessageBoxW");
        NakedByteStream nbs = new NakedByteStream();
        LittleEndianOutputStream leos = new
            LittleEndianOutputStream(nbs);
        leos.writeStringUnicode(msg);
        leos.writeStringUnicode(title);
    }
}
```

```

        GenericStub.win32Invoke(fp.getPeer(),
                                "kGGk:T1:",
                                mstackMessageBoxW,
                                leos.size(),
                                nbs.getInternalBuffer(),
                                null);
    }
    //etc.
}

```

The `User32` stub for `MessageBoxW` executes the following steps:

1. Create an instance of the `FuncPtr` helper class. Behind the scenes, this helper calls `LoadLibrary` and `GetProcAddress` to load the function.
2. Create a `LittleEndianOutputStream` to hold the request message. (Win32 and COM expect bytes to be in little-endian order).
3. Write the string arguments into the stream.
4. Call the method, passing in the request and the instruction string.

The instruction string `kGGk:T1:` is interpreted as follows:

- The characters before the first colon are the instructions for converting the message into a call buffer. The letter `k` means “skip this argument on the stack” and the letter `G` means “read a string from the message and write its address onto the stack.” In this example, the first and last arguments are skipped because the entire call buffer is zero-filled, and these arguments need to be zero.
- The characters between the colons are the instructions for writing the return value into the response buffer. The characters `T1` mean “write the return value into the response buffer *and* raise a `COMException` if the function returned zero.”
- The characters after the second colon are instructions for writing any `out` parameters into the response buffer. `MessageBoxW` does not have any `out` parameters, so this part of the string is empty.

Jawin supports a large number of different instructions which are documented at [Jawin] and can be extended to support arbitrarily complex API signatures.

Jawin’s character encodings for the instruction strings are arbitrary and can be quite complex. The purpose of the generic stub is to replace raw JNI for complex method signatures. However, one could make the argument that learning to

use encodings such as `kGGk:T1` is just as difficult as writing raw JNI, so why bother? The answer, of course, is that you will not be writing the encodings directly. Classes such as `User32` should be generated from type information, as discussed in the next section.

## A.7 Generating Stubs

Consider again the marshalling architecture diagram, shown in Figure A-1. The intrinsic functions and other marshalling infrastructure code must be developed by hand. However, they are developed only once to be shared by all interface stubs. Each particular COM interface or set of DLL entry points will need its own interface stub. These interface stubs vary only by differences in type information, and they are ideal candidates for code generation. Jawin includes prototypes for generating both shared stubs and interface stubs.

### A.7.1 Generating Shared Stubs

The `com.develop.jawin.tools` package, included with Jawin, is a pure Java implementation that builds the Java and native source code files for shared stubs. The `COMSharedStubDriver` class defines a set of argument types and a maximum number of arguments to generate. Then the `COMSharedStubDriver` iterates over every permutation of arguments, calling `COMSharedStubBuilder` to generate the Java declaration and native implementation. Listing A-9 shows the native stub declarations generated for two-argument methods with types `int`, `float` and `String`. Listing A-10 shows a small sample of the generated implementations.

#### Listing A-9 Stub Declarations Generated by `COMSharedStubDriver`

```
//stub declarations
package com.develop.com.marshall;import com.develop.com.*;import
com.develop.util.*;import java.io.*;import java.util.*;
    public class COMMarshal {    public static native void
        invokeII(int vtableIndex, int guidToken, int peer, int
        unknown, int arg0, int arg1);
    public static native void invokeIF(int vtableIndex, int
        guidToken, int peer, int unknown, int arg0, float arg1);
```

```

public static native void invokeIG(int vtableIndex, int
    guidToken, int peer, int unknown, int arg0, String arg1);
public static native String invokeIoG(int vtableIndex, int
    guidToken, int peer, int unknown, int arg0);
public static native int invokeIoI(int vtableIndex, int
    guidToken, int peer, int unknown, int arg0);
public static native float invokeIoF(int vtableIndex, int
    guidToken, int peer, int unknown, int arg0);
public static native void invokeFI(int vtableIndex, int
    guidToken, int peer, int unknown, float arg0, int arg1);
public static native void invokeFF(int vtableIndex, int
    guidToken, int peer, int unknown, float arg0, float arg1);
public static native void invokeFG(int vtableIndex, int
    guidToken, int peer, int unknown, float arg0, String arg1);
public static native String invokeFoG(int vtableIndex, int
    guidToken, int peer, int unknown, float arg0);
public static native int invokeFoI(int vtableIndex, int
    guidToken, int peer, int unknown, float arg0);
public static native float invokeFoF(int vtableIndex, int
    guidToken, int peer, int unknown, float arg0);
public static native void invokeGI(int vtableIndex, int
    guidToken, int peer, int unknown, String arg0, int arg1);
public static native void invokeGF(int vtableIndex, int
    guidToken, int peer, int unknown, String arg0, float arg1);
public static native void invokeGG(int vtableIndex, int
    guidToken, int peer, int unknown, String arg0, String arg1);
public static native String invokeGoG(int vtableIndex, int
    guidToken, int peer, int unknown, String arg0);
public static native int invokeGoI(int vtableIndex, int
    guidToken, int peer, int unknown, String arg0);
public static native float invokeGoF(int vtableIndex, int
    guidToken, int peer, int unknown, String arg0);
public static native void invokeI(int vtableIndex, int
    guidToken, int peer, int unknown, int arg0);
public static native void invokeF(int vtableIndex, int
    guidToken, int peer, int unknown, float arg0);
public static native void invokeG(int vtableIndex, int
    guidToken, int peer, int unknown, String arg0);
public static native String invokeeoG(int vtableIndex, int
    guidToken, int peer, int unknown);
public static native int invokeeoI(int vtableIndex, int
    guidToken, int peer, int unknown);
public static native float invokeeoF(int vtableIndex, int
    guidToken, int peer, int unknown);
}

```

**Listing A-10 Native Stubs Generated by COMSharedStubDriver**

```

JNIEXPORT void JNICALL
Java_com_develop_com_marshal_COMMarshal_invokeIF(JNIComUtil *
pEnv, jclass, jint vtableIndex, jint guidToken, jint peer, jint
unknown, jint arg0, jfloat arg1)
{
    CComPtr<IUnknown> cpUnk;
    try {
        getUnknown(guidToken, peer, unknown, &cpUnk);
        FTYPE3* vtable = (FTYPE3*) (*(int*)(cpUnk.p));
        int inv0 = arg0;
        float inv1 = arg1;
        JNI_HR(vtable[vtableIndex]((int)cpUnk.p, inv0, inv1));
    }
    HANDLE_JNI_EXCEPTIONS()
}

JNIEXPORT void JNICALL
Java_com_develop_com_marshal_COMMarshal_invokeIG(JNIComUtil *
pEnv, jclass, jint vtableIndex, jint guidToken, jint peer, jint
unknown, jint arg0, jstring arg1)
{
    CComPtr<IUnknown> cpUnk;
    try {
        getUnknown(guidToken, peer, unknown, &cpUnk);
        FTYPE3* vtable = (FTYPE3*) (*(int*)(cpUnk.p));
        int inv0 = arg0;
        CComBSTR temp1;
        temp1.Attach(pEnv->jstobs(arg1));
        int inv1 = (int)temp1.m_str;
        JNI_HR(vtable[vtableIndex]((int)cpUnk.p, inv0, inv1));
    }
    HANDLE_JNI_EXCEPTIONS()
}

```

The code for the actual generator is almost insultingly simple. That is one of the beauties of generative programming. There is often no need (or temptation) to optimize the generator because it is run only at development time. Also, the generator does not need to be particularly user-friendly since the target user is a developer. Of course, a very large, widely used generator might need to be developed and optimized like a “normal” program. Nevertheless, generators are often simpler than the same logic implemented dynamically at runtime.

One reason that the generator is simple is that “almost right is good enough.” The `COMSharedStubDriver` makes several simplifying assumptions about `out` parameters:

- There is at most one `out` parameter per method.
- If there is an `out` parameter, it appears last.
- The `out` parameters should be translated to a Java return value.

None of these assumptions is entirely true, but they probably apply to better than 98 percent of actual COM interfaces. Many developers would consider it unwise to code these assumptions into an OO base class since they are not entirely accurate and might compromise the architecture. Nobody worries if a development-time generator is predictably inaccurate for some cases. You can always replace the generator with another generator, defer special classes to another generator, or simply code the outlying cases by hand.<sup>6</sup>

### A.7.2 Generating Interface Stubs

Whether you use shared stubs or a generic stub, the front end of the marshalling layer is an interface stub that provides a Java representation of some COM interface or Win32 entry point. In general, the source code for an interface stub looks like this:

```
//pseudo-code
package <% =some.arbitrary.package %>;
import <%= some.standard.imports %>;

class <%= SomeStub %> {
    //Some per-class goo...

    <%= SomeReturnType %>
    <%= someMethod %>(<%= someArgs %>) { //for each method
        SomeCallStackRep r = new SomeCallStackRep();
        r.addArg(<%= arg[n] %>); //for each arg
```

---

6. This entire argument is as much about culture as it is about technology. It is entirely feasible to code an inheritance-based solution that deliberately ignores special cases. However, many OO purists are obsessive about accurately modeling the problem domain. The recent rise of Extreme Programming (XP) is, in part, a rejection of this aspect of OO culture.

```

        ReturnRep ret = Stub.invoke(r);
        return <%= someReturnTypeConversionFunc %>(ret);
    }
}

```

Most of the text in this file is boilerplate; only the bolded portions need to change for different methods. Type information can be used to generate these replacements. The use of the `<%= expr%>` syntax implies an obvious approach, which is to use JSP or a JSP-like syntax to generate the source code for the stubs.

Because COM type information is already available through COM interfaces, Jawin uses a COM-based code generation language called X-Code<sup>7</sup> instead of JSP. Listing A-11 shows the method generation portion of the X-Code template, and Listing A-12 shows an example of a generated method.

#### Listing A-11 Method Generation Template from COMThunk.xjava

```

public <%= returnTypeNames(method) %>
<%= method.Name %>(<%= argList(method,0) %>)
throws COMException, IOException
{
    int vtIndex = <%= method.VtableIndex %>;
    NakedByteStream nbs = new NakedByteStream();
    LittleEndianOutputStream leos =
        new LittleEndianOutputStream(nbs);
    //arg stream
    <% for (i=0; i < method.ArgCount; i++) { %>
    <%= marshalArg(method, i)%><% } %>
    //object args
    <%= customObjArray(method) %>
    <% for (i=0; i < method.ArgCount; i++) { %>
    <%= marshalObject(method, i)%><% } %>
    byte[] result = GenericStub.comInvokeString(
        "<%= marshalString(method) %>",
        <%= method.ArgCount * 4 %>,
        leos.size(),
        nbs.getInternalBuffer(),

```

7. X-Code is part of Gen<X>, a commercial code general reaction tool developed by the author's employer. Jawin began as a proof of concept for X-Code, and Jawin itself is completely free and open source. For more information on Gen<X>, see [GenX].

```

        objArgs,
        vtIndex,
        iidToken,
        peer,
        unknown);
    <%= customMaybeStream() %><%= customMaybeReturn() %>
}

```

### Listing A-12 Sample Generated Method: Saving a Word Document

```

public void Save(Object arg0, Object arg1)
    throws COMException, IOException
{
    int vtIndex = 16;
    NakedByteStream nbs = new NakedByteStream();
    LittleEndianOutputStream leos =
        new LittleEndianOutputStream(nbs);
    //arg stream
    Variant.marshalIn(arg0, leos);
    Variant.marshalIn(arg1, leos);
    //object args
    Object[] objArgs=null;

    byte[] result = GenericStub.comInvokeString("VV:H:11",
        8,
        leos.size(),
        nbs.getInternalBuffer(),
        objArgs,
        vtIndex,
        iidToken,
        peer,
        unknown);
}

```

The `Save` method implementation shows both the benefits and limits of a type-information driven approach. On the plus side, the generation of this method is entirely automatic; all you have to do is run the template against the Microsoft Word type library. However, the signature is not very informative. Many COM interfaces are designed primarily with scripting in mind, and they do not use strong typing. The `Save` method uses `VARIANT` arguments, which can be any legal scripting type. As a result, the Java stub is weakly typed as well, taking arguments of type `Object`.





## A.8 Onward

Interoperation between Java, Win32, and COM is an important aspect of most enterprise systems. Even organizations with a strong commitment to Java need access to the huge number of existing COM and Win32 DLL components.

There is no single solution to this problem, and the relationship between the key vendors makes it unlikely that there ever will be. [JavaWin32] summarizes the various third-party products that have appeared to fill this void. To varying degrees, each product hides the sources of platform impedance presented in §A.3.

The latter half of this appendix presented an example marshalling architecture. You have three important decisions to make in choosing the best marshalling architecture for your own use:

1. Decide which aspects of platform impedance need to be handled transparently, which ones to expose to Java clients, and how they should be exposed.
2. Find support for the type systems you need. If you only need to access scriptable COM components, there is no need to invest in a product that supports DLL entry points or complex IDL expressions.
3. Minimize the work that must be done by hand. Choose a solution that can automatically generate the interface stubs you need.

Many of the marshalling products will let you mix-and-match solutions to these three questions. For example, since Jawin is open source, you could easily use Jawin's intrinsic functions but replace its stub generators with your own.

This appendix has examined in-process communication. In-process solutions have several benefits. They are almost certain to be faster because they do not pay the penalty of crossing a process or network boundary. Also, they allow Java programs direct access to process-local resources. If you need access to process-local resources, in-process interop is your only choice.

Another possibility, beyond the scope of this book, is out-of-process techniques, using network communication to bridge between Java and native processes. Calling native code in another process provides some fault tolerance since the native process has no way to damage the Java process. On the other hand, out-of-process communication is slower and is completely unsuitable for some tasks, such as manipulating process-specific resources inside the virtual machine.



