
JADE Development Centre

External Interfaces



In this paper . . .

INTRODUCTION	1
TYPE OF EXTERNAL INTERFACE	2
USING AN EXTERNAL PROCESS	3
JADE-SIDE OF EXTERNAL FUNCTION OR METHOD	7
BUILD ENVIRONMENT AND COMPILER SETUP	10
WRITING AN EXTERNAL FUNCTION	14
WRITING AN EXTERNAL METHOD	16
DESIGN AND CODING ISSUES	20
SUMMARY	22

Introduction

The JADE RootSchema classes provide significant functionality but where one needs to interface to either a legacy system or the operating system the use of external interfaces may be required. This white paper focuses on the API level external interfaces that JADE provides, and how to use them.

Guidelines are presented for selecting the best external interface to use for differing requirements. Samples show how to build and interface to external code and points out implementation issues such as portability, multi-language support, and thread safety.

Other forms of external interfaces that are not covered in this white paper are ODBC, TCP/IP, Named Pipes, and ActiveX.

Type of External Interface

JADE provides three types of external Application Programming Interface (API).

1) Initiate an external program. 2) Call to an external function passing simple primitive data types. 3) Call an external method that can accept both primitive data types as well as JADE objects. As you are dealing with external interfaces, you need to be aware of and handle operating system, installation, and environmental differences.

External Process

The external process interface is useful where the operation required is currently supplied in a discreet program or script. This program can be started on the computer running the JADE database server, application server, standard client, or a smart thin client. Starting the external process is achieved by using the JADE **Node::createExternalProcess** method. The standard way of passing information to the external process from JADE is via the command line or writing to files, and to retrieve results from the external process is by the process exit code or reading from files.

External Function

External functions are native code that provides a C language API. They are often provided as a third-party library, or possibly written by you. They can be called on the computer running the JADE database server, application server, standard client, or a smart thin client. Once the function has been defined to JADE, it is available through the use of the **call** statement. Parameters to the function can be passed as input, output, or input-output; the restriction is that the type of a parameter must be a JADE primitive type.

Use external functions when the code must run on a smart thin client. Also they are an ideal choice when used as wrappers to a third-party library that only requires simple primitive types.

External Method

The external method is similar to the external function in that it is native code that provides a C language API. It differs in that all external methods have the same fixed format C-API, and the JADE Object Manager is required to handle the parameters. External methods normally need to be written by you or as part of a schema in the hierarchy.

They can be called on the computer running the JADE database server, application server, standard client, but not from the smart thin client. The restriction is that it is unavailable to smart thin clients because the JADE Object Manager is not part of the smart thin client. The library containing one or more external methods is defined to JADE and the actual external method is associated with a primitive or class method. Using this external method is now no different to using a method

written in the JADE language. Parameters to the method can be of any primitive type or JADE object and passed in any supported manner.

External methods are best for when third-party libraries have complex arguments such as C structures or enumerated types. This allows objects to be passed across the external method interface. This avoids the use of the JADE binary primitive type that has potential byte ordering and alignment issues. Also any state requirements can be maintained in the JADE object.

Using an External Process

Care needs to be taken with respect to terminology used, as the word *process* has different meanings depending on the context. A JADE process is basically one thread within a JADE node. The JADE *node* is an operating system process.

The JADE method to create an external process has the following signature.

```
Node::createExternalProcess(  
    directory: String;  
    command: String;  
    args: StringArray;  
    alias: String;  
    thinClient: Boolean;  
    modal: Boolean;  
    result: Integer output): Integer
```

Figure 1

From the JADE perspective, the external process is a single task that can either be set running and ignored (**modal** = false), or the JADE process stops until the external process completes and the results are checked (**modal** = true). There generally is no interaction with the external process while it is executing.

The JADE programmer needs to be aware where the external process is going to execute and what arguments the external process requires. Code that is intended to be portable and resilient needs to deal with a number of situations, even for the apparently simple task of starting the Microsoft utility WordPad.

Setting the **thinClient** parameter of **createExternalProcess** to false works for standard clients, but is incorrect when the method runs on node that is executing on the AIX operating system, and is unusual if the node is an application or database server. Forcing the execution onto the thin client makes sense for something like WordPad, but the code still needs to deal with the situations where the external application has not been installed or is not in a location that Windows expects to find it.

Example: Overview

The following example could be used as part of a server application where a JADE database and application server have been installed as services. The database must be up and running before the application servers can be started. The method uses **createExternalProcess** to execute the correct external operating system specific command to start or stop the application server service.

Microsoft Windows NT and 2000 have the **net start "service name"** and **net stop "service name"** commands to control services. The JADE product on AIX provides the commands **jadestartsrc -s "service name"** and **jadestopsrc -s "service name"** to provide the similar functionality.

```

netService(start : Boolean; name : String) : String serverExecution;

vars
    command, alias, version      : String;
    arguments                    : StringArray;
    methodResult, processResult  : Integer;
    osType, architecture        : Integer;

begin
    create arguments transient;
    osType := node.getOSPlatform( version, architecture );
    if osType.bitAnd(Node.OSWindows) <> 0 then
        if osType = Node.OSWindowsHome then
            return "Not support for Services";
        endif;
        command := 'net';
        alias := null;
        if start then
            arguments.add('start');
        else
            arguments.add('stop');
        endif;
        arguments.add('"' & name & '"');
    elseif osType.bitAnd(Node.OSUnix) <> 0 then
        command := '/usr/jade/sbin/jadecontrol';
        if start then
            alias := 'jadestartsrc';
        else
            alias := 'jadestopsrc';
        endif;
        arguments.add('-s');
        arguments.add(name);
    endif;

    methodResult := node.createExternalProcess( ".",
                                                command, arguments, alias,
                                                false, true, processResult );

    if methodResult = Node.ExternalProcess_Successful then
        if processResult = 0 then
            return "OK";
        endif;
    endif;
end

```

```
        else
            return "External Process returned " &
                processResult.String;
        endif;
    else
        return "Problem with method createExternalProcess";
    endif;
epilog
    delete arguments;
end;
```

Figure 2

Example: Description of Code

To allow this method to be portable between Windows and AIX, you need to check which operating system this method is currently executing on and then deal with issues specific to each platform.

- If the operating system type is Windows, ensure that the functionality we are about to use is installed. In the case of the home (consumer) (Windows 9x, Windows ME, and so on) versions of Windows services are not implemented.
- As the **net** command is normally located in the **system32** subdirectory of the Windows install directory, it is expected the operating system will be able to locate the command. For this reason, the full path to the **net** command has not been given.
- You need to take care to put quote marks around the service name just in case it contains embedded spaces. This is because on Windows the arguments to the command are passed as a single string and it is the command's responsibility to break apart the individual arguments.

If the operating system is UNIX (as of JADE 5.2, the only supported UNIX implementation is AIX), the following applies.

- We set the command to a full path name. This has been done for two complimentary reasons. Firstly, the JADE install process puts this program in a fixed location, and secondly, the server may not have been started with the **PATH** environment variable set to search the directory where **jadecontrol** program has been installed.
- As this program can be known by several names and will perform different actions depending on how has been called, we need to set the **alias** parameter to the name of the desired command (the value of **argv[0]** in C programmer terminology). The ability to give one program on disk multiple names is a feature of the UNIX operating system. The example uses the **alias** parameter to indicate the desired name to be used. As an alternate implementation, the **command** parameter could have been set to either **/usr/jade/sbin/jadestartsrc** or **/usr/jade/sbin/jadestopsrc** and the alias set to null.
- The service name argument does not need to be quoted, as the individual arguments are passed to the external command being executed.

When all of the operating system differences have been allowed for, the external process is executed. The external process is initiated in a modal mode because we

want to get a status result back from the command. Both commands in this example return zero (0) on success, but the failure return values are different. If desired, the method could be enhanced to handle the different failure return values and also capture any text output messages from these commands. Adding support to capture the text output from the external command involves two of steps. The first is getting the command output into a file and the second using JADE logic to read the text file.

The following code snippet shows the first step. To get a command's standard output into a text file often requires invoking the operating system's command processor with the name of the primary command and its arguments, along with additional command processor arguments needed to redirect the output. The normal command processor for Microsoft Windows NT/2000 is **cmd.exe** and for UNIX systems **/bin/sh**.

```

osType := node.getOSPlatform( version, architecture );
if osType.bitAnd(Node.OSWindows) <> 0 then
    // cmd /c "net start "service name" > extproc_2.txt
    command := "cmd.exe";
    logfile := "..\logs\extproc_" &
                process.number.String & ".txt";
    if start then
        arguments.add('/c "net start "' & name &
                        '"" > ' & logfile);
    else
        arguments.add('/c "net stop "' & name &
                        '"" > ' & logfile);
    endif;
elseif osType.bitAnd(Node.OSUnix) <> 0 then
    // sh -c 'jadestartsrc -s "service name" > extproc_2.txt
    command := '/bin/sh';
    arguments.add('-c');
    if start then
        arguments.add('/usr/jade/sbin/jadestartsrc -s
                        "' & name & "'');
    else
        arguments.add('/usr/jade/sbin/jadestopsrc -s
                        "' & name & "'');
    endif;
    arguments.add('>');
    logfile := "../logs/extproc_" &
                process.number.String & ".txt";
    arguments.add(logfile);
endif;

methodResult := node.createExternalProcess(
                    app.getJadeInstallDir, command,
                    arguments, null,
                    false, true, processResult );

```

Figure 3

The variable **logfile** includes the JADE process number to help with uniqueness. You could also consider including the JADE node number, depending on application requirements. The directory from which we execute the external process is now important as we supplied a relative path name for the output file. Due to the change in quoting required by the command processor, the code looks quite different but the same basic structure has been kept. The comments show a sample of the command line we are producing in the JADE code.

See “Design and Coding Issues” later in this document, for more information on handling the environmental and operating system differences.

Tip: You could write a wrapper script as your external process and place it in the JADE **bin** directory. This way the JADE logic is simplified, as you will always know where the external script is located and the script can handle the operating system and installation differences. But on the down side, this script needs to be distributed with each JADE installation

JADE-Side of External Function or Method

In the following sections on external functions and methods, the same basic example is used. Compare two strings to see if they are equal, using a number of different interface techniques.

JADE requires that both external functions and external methods provide their interfaces as C language APIs in dynamically loadable libraries. Both Microsoft Windows and UNIX support this facility. The actual language that the external code is written in is your choice.

The first step is to define the name of your native library in the JADE Library Browser window.

Next, if you are implementing an external function, use the JADE External Functions Browser window to add your external function. In the dialog, it allows you to specify the name for your function (**isEqual**). This is the name your JADE code will use. Also specify the entry point (**function_compare**), which is the name exported from your library. External function names must be unique across all external libraries. An initial function signature is provided and this needs to be changed to match what your entry point actually requires (**s1 : String; s2 : String) : Boolean**).

When dealing with strings, JADE handles the differences between ANSI and Unicode for you. However ANSI and Unicode strings at the C level they are different, so your External Function code needs to deal with this, as the C function signatures are different.

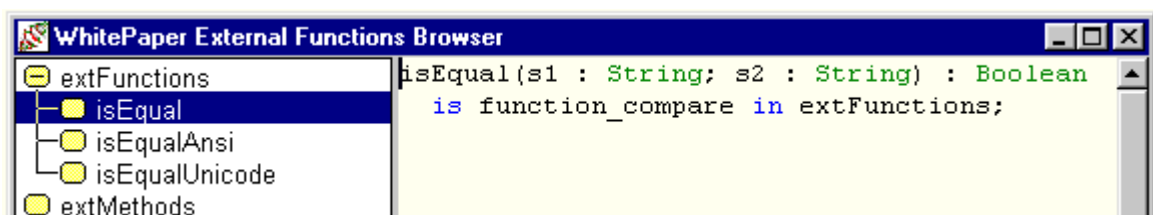


Figure 4

As a design choice, you can implement a single function to handle both ANSI and Unicode versions of JADE. This requires that you compile your library twice and ensure the correct one (ANSI or Unicode) is installed with the same build of JADE. (See Figure 4.)

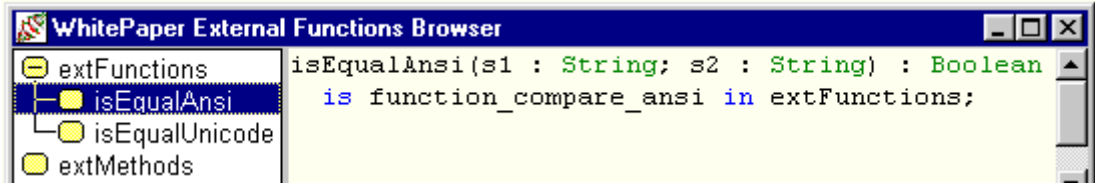


Figure 5

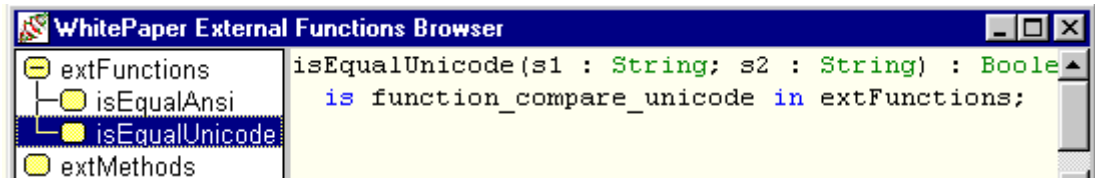


Figure 6

Alternatively, you can define two entry points in your library: one for each of the ANSI and Unicode modes that JADE runs in. This library only needs to be compiled once and then can be shipped with either JADE build. (See Figure 5 and Figure 6.)

```
vars
  s1, s2 : String;
  same : Boolean;
begin
  ...
  if app.isUnicode then
    same := call isEqualUnicode(s1, s2);
  else
    same := call isEqualAnsi(s1, s2);
  endif;
  ...
end
```

Figure 7

Figure 7 shows a fragment of JADE code that allows your code to select the correct entry point into a library when it has both the ANSI and Unicode implementations of a function.

Next if you are implementing an external method, go to your selected primitive type or class and use the New External Method menu item that brings up the External Method Definition dialog to add your new external function. In the dialog, it allows you to specify the name of the method (**isEqual**), which is the name your JADE code uses. In addition, specify the entry point (**primitive_compare**), which is the name exported from your library (**extMethods**). External method names must be unique within the class. A default method signature is provided, but this needs to be changed to match your requirements.

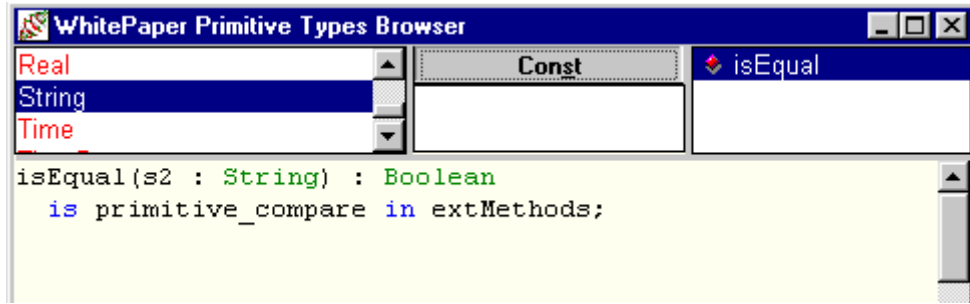


Figure 8

Figure 8. Illustrates an external method added to a String primitive type, so that the contents of primitive can be compared to the *s2* parameter.

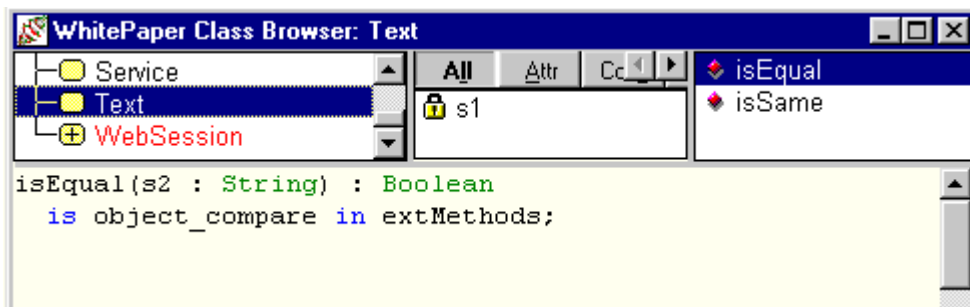


Figure 9

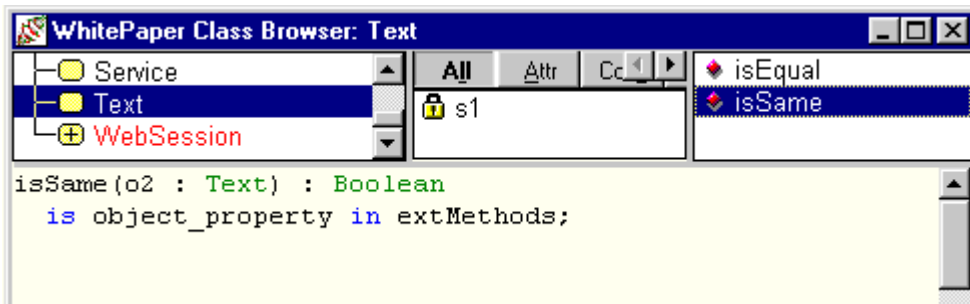


Figure 10

The above figures show adding external methods to a class. Figure 9 accepts a primitive type, so the *s2* String parameter is compared to the *s1* string property. Figure 10 takes an object of the same type, allowing the comparison of *self.s1* with *o2.s1*.

JADE does not put any restrictions on having external functions and methods in the same library but be aware that if the library is to be used on a JADE smart thin client, there will be runtime linking issues when JADE loads the library into memory unless you have gone to the effort of making all JOM C-API entry points dynamically loaded. Therefore it is recommended that external functions and methods be kept in separate libraries, unless the usage of this support library is known in advance.

When an external function or method is called, JADE checks to see if the library containing the native code has been loaded. If necessary the library is automatically loaded. Then, as an entry point is required, its address in the library is resolved. The library stays loaded until the JADE node, or smart thin client, terminates.

If the library is not located, the JADE exception 1006 **Library Not Found** is raised. Given that the library is loaded successfully, the other JADE exception that your code may have to deal with is 1008, **Entry Point Not In Library**.

Warning: JADE is not able to check that the C-API function signature you supply is correct. JADE does its best to protect against possible errors, but if it is incorrect then at best you can expect unpredictable results but more likely an application crash occurs. Therefore ensure the supplied signature accurately matches the one in the actual C API.

Note: The external library needs to be locatable by the standard operating system search method. If you have control over the library installation, it is recommended that it be placed in the JADE **bin** directory on Windows or the **lib** directory on AIX.

Build Environment and Compiler Setup

Documentation and Samples

The standard JADE documentation provides information about developing external interfaces. To understand more about these topics review the following documentation:

- *JADE Developers Reference*, Chapter 4, "Using External Methods and External Functions"
- *JADE Object Manager Guide*, Chapter 3, "JADE Application Programming Interface (API)"

Sample external interface C source code is provided in the **examples/demodll** directory, providing source code, Microsoft project files, and a UNIX **makefile**.

Microsoft Windows Environment

To set up the Microsoft Visual C/C++ version 6.0 compiler, the following main points should be noted.

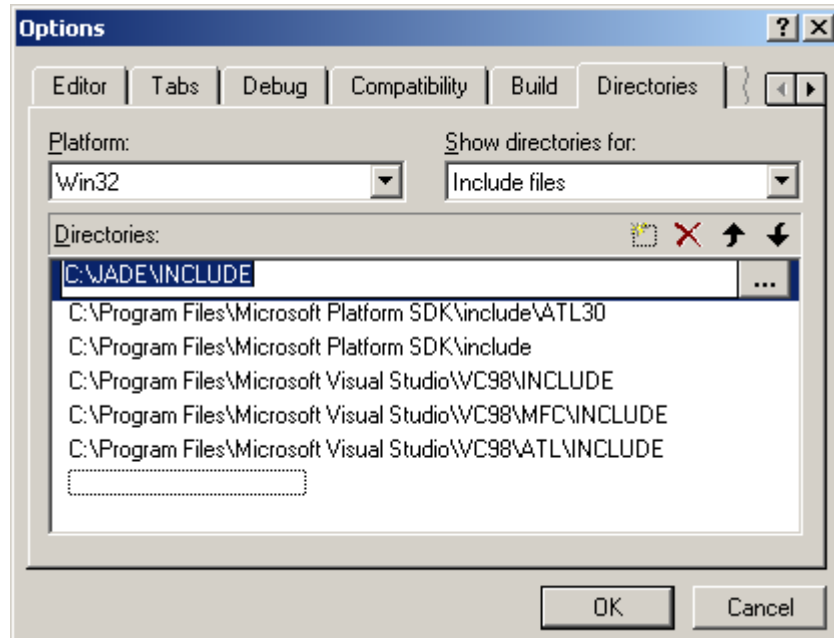


Figure 11

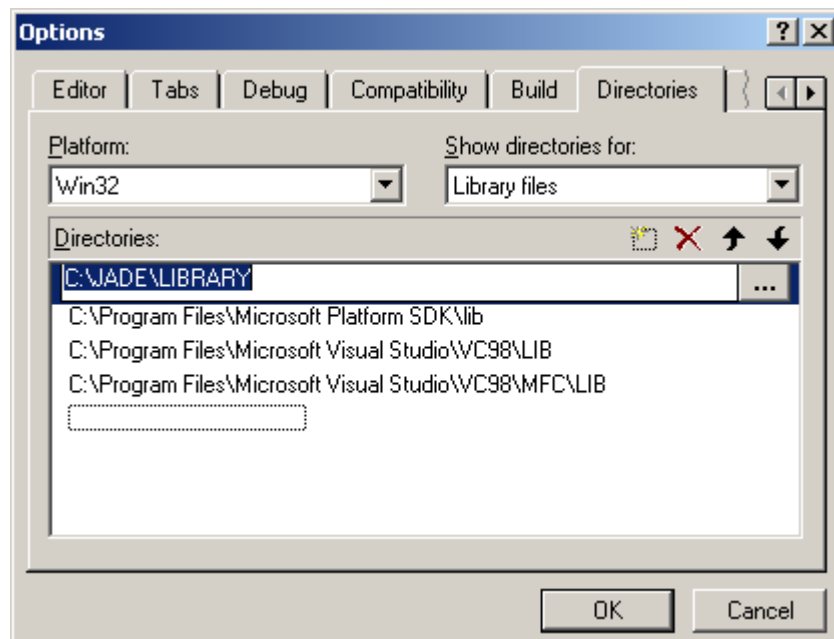


Figure 12

Figure 11 and Figure 12 tell the Microsoft compiler where to find the header files and library files, respectively. These settings are on a development environment level, rather than relating to a specific project. If you are developing external code for multiple versions of JADE, be aware that you need to recompile your code against the specific version of the JADE object manager libraries.

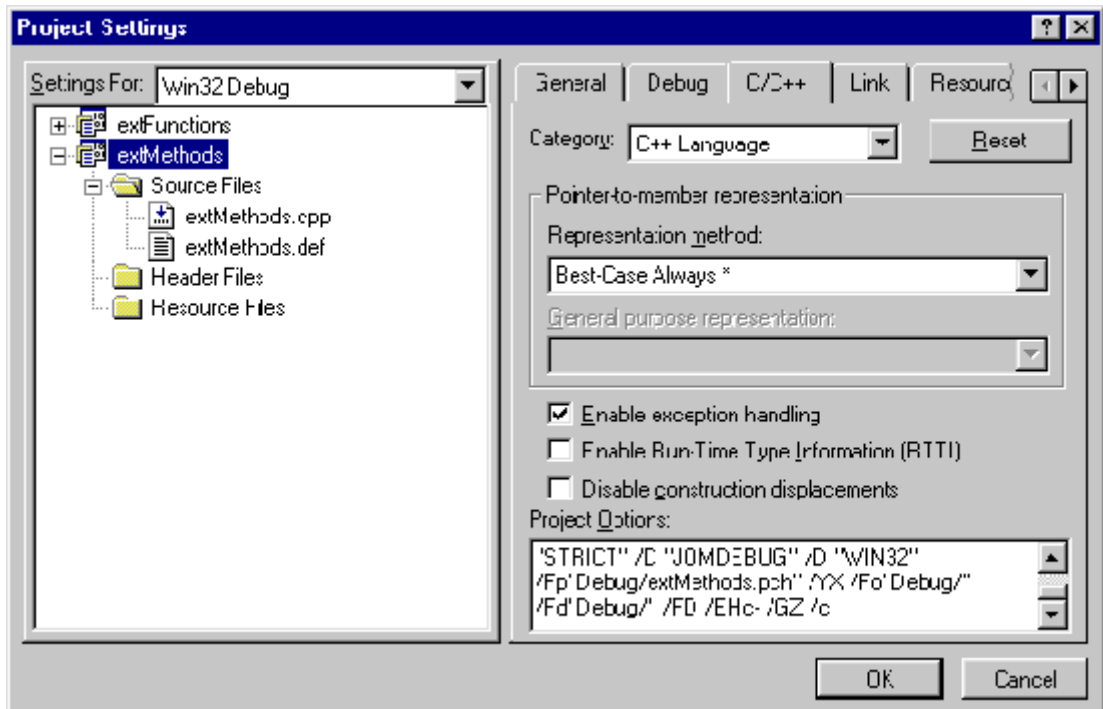


Figure 13

Figure 13 shows the settings needed to support C/C++ exception handling in an external function or method. Also to get the compiler to perform the required type of exception handling the **/EHc-** flag has been added.

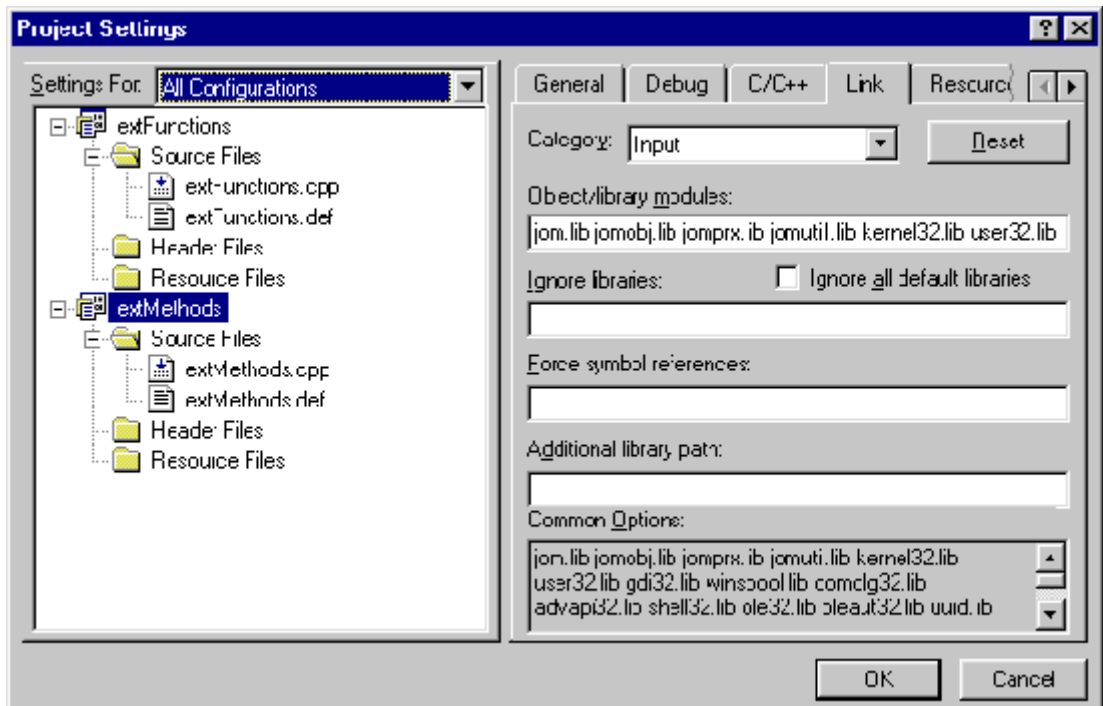


Figure 14

Figure 14 lists the libraries that C/C++ needs to link against if the JADE object manager C-API is used.

IBM AIX Environment

To build external dynamic libraries on AIX requires the IBM Visual Age Compiler version 5.0 (as from JADE version 5.2).

The following three figures (Figure 15, Figure 16, and Figure 17) combine to create a standard UNIX **makefile**. This example only shows the **makefile** for the **extMethods** library, but the **extFunctions** **makefile** differs only in the file names used.

```
## Setup variables and options

# Debug version
COPTIM      = -g -qdbxextra
LDOPTIM     =
# Release version
#LDOPTIM    = -s
#COPTIM     = -O3 -qstrict
# Unicode version
#UNICODE    = -DUNICODE

# Directories
VACDIR      = /usr/vacpp
TOPDIR     = ../../..
```

Figure 15

The variables block of the **makefile** (Figure 15) allows you to select how the code should be compiled, either debug or release. The variable **TOPDIR** gives a relative path from the location where the source files exist to the top of the JADE **install** directory.

```
## Build flags and tools

# Compiler Flags
CC          = $(VACDIR)/bin/xlC_r
XLCFLAGS    = -qlonglong -qlongdouble -qarch=com \
              -qrtti=all -qro -qroconst -qfuncsect \
              -qenum=int -qinfo=por:nound
INCLUDES    = -I$(TOPDIR)/include
DEFINES     = -DUNIX $(UNICODE)
CFLAGS      = $(XLCFLAGS) $(COPTIM) \
              $(INCLUDES) $(DEFINES)

# IBM C++ Shared library support
MKSHLIB     = $(VACDIR)/bin/makeC++SharedLib_r

# Linker/binder
LIBS        = -lm
LD          = $(CC)
LDFLAGS     = $(LDOPTIM) -L $(TOPDIR)/lib \
              $(TOPDIR)/lib/jade.imp -brtl -bnodelcsect
```

Figure 16

The **build flags** section of the **makefile** (Figure 16) set up the correct compiler and linker flags. The Visual Age compiler is available under a number of different names on disk. The name controls some significant compiler options; the name **xlC_r** indicates that this is the C++ compiler supporting multi-threaded applications.

```
## Dependencies and targets
```

```

ALL : extMethods.so extMethods.a

extMethods.o : extMethods.cpp
              $(CC) $(CFLAGS) -c extMethods.cpp

shr.o : extMethods.o
        $(MKSHLIB) $(LDFLAGS) -bl:shr.map -o shr.o \
        -p 0 extMethods.o

extMethods.a : shr.o
              /usr/css/bin/ar crv extMethods.a shr.o

extMethods.so : extMethods.o
               $(MKSHLIB) $(LDFLAGS) -bl:extMethods.map \
               -o extMethods.so -p 0 extMethods.o $(LIBS)

```

Figure 17

The rules section of the **makefile** (Figure 17) tells the **make** program which source files produce which objects, and which object files combine together to produce the dynamic library. AIX library archives (**.a** files) are different to most other UNIX versions as they concurrently support both statically linked and dynamic loaded objects, which is the reason for the extra link step to produce **shr.o**. For more details see the IBM “General Programming Concepts: Writing and Debugging Programs“ documentation under “Creating a Shared Library”.

Writing an External Function

The following four figures (Figure 18, Figure 19, Figure 20, and Figure 21) make up the **extFunctions.cpp** file, they are broken down into several figures to aid discussion.

```

#if defined(UNIX)
#include <pthread.h>    // AIX: REQUIRES this to be FIRST
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#else /* otherwise must be windows */
#include <windows.h>
#endif /* defined(UNIX) */
#include "jomtypes.h"
#include <string.h>

```

Figure 18

The **include** file directives (Figure 18) come in three groups that are specific to UNIX, Windows, or common to both environments. Other available defines that can be used are **_Windows** (provided by **jomarch.h**) and **_AIX** (provided by the Visual Age C++ compiler).

```

/* Single function, library per ANSI and Unicode */
extern "C" int JOMAPI
function_compare(
    const Character * s1, const Character * s2 )
{
    /* For Windows only hosts */
    #if defined(_Windows)
        return lstrcmp( s1, s2 ) == 0 ? TRUE : FALSE;
    #else

```

```

        /* Works on both Windows and UNIX */
#if defined(UNICODE)
    return wcscmp( s1, s2 ) == 0 ? TRUE : FALSE;
#else
    return strcmp( s1, s2 ) == 0 ? TRUE : FALSE;
#endif
#endif
}

```

Figure 19

The **function_compare** entry point (Figure 19) shows how you could write a single function to support Windows and UNIX as well as ANSI and Unicode runtime modes, depending how and where the source file was compiled. The Character type (defined in **jomarch.h**) is the correct type depending on Unicode being defined or not. The function **lstrcmp** is commonly used in Windows to allow dynamic selection between ANSI and Unicode builds. The standard C language **strcmp** and **wcscmp** functions are also implemented on Windows. If portable code is required, then it is recommended that you write to the specifications defined by standards bodies rather than using compiler vendor enhancements. This also aids in producing consistent behavior across operating systems.

Note: All of the provided sample code makes the assumption that the strings passed from JADE to the C/C++ code do not have any embedded NULL characters. A major difference between the standard C and C++ libraries and JADE is the interpretation of NULL characters. JADE allows embedded NULL characters to be part of a string whereas C/C++ take the first NULL character to be the string terminator.

```

/* Windows and UNIX, single library for both ANSI and Unicode */
extern "C" int JOMAPI
function_compare_ansi(
    const char * s1, const char * s2 )
{
    return strcmp( s1, s2 ) == 0 ? TRUE : FALSE;
}

```

Figure 20

```

extern "C" int JOMAPI
function_compare_unicode(
    const wchar_t * s1, const wchar_t * s2 )
{
    return wcscmp( s1, s2 ) == 0 ? TRUE : FALSE;
}

```

Figure 21

The **function_compare_ansi** (Figure 20) and **function_compare_unicode** (Figure 21) are cleaner functions than **function_compare** (Figure 19), because they avoid the use of conditional compile directives. The main differences are the use of standard C types **char** for ANSI and **wchar_t** for wide character Unicode, as well as using non-vendor-specific functions.

```

LIBRARY extFunctions

DESCRIPTION 'WhitePaper External Functions'

EXPORTS
    function_compare
    function_compare_ansi
    function_compare_unicode

```

Figure 22

The definition file (Figure 22) is specific to the Microsoft environment. Defining a function as **extern "C" return_type JOMAPI** is not sufficient to get a correctly exported library entry point suitable for JADE. You must also specify the function names in a definition file before the JADE object manager can load these entry points.

Writing an External Method

The following six figures (Figure 23, Figure 24, Figure 25, Figure 26, Figure 27, and Figure 28) make up the **extMethods.cpp** file. This file implements the **primitive_compare**, **object_compare**, and **object_property** external methods. They are broken down into several figures to aid discussion.

```
// Application required includes
#if defined(UNIX)
#include <pthread.h>    // AIX: REQUIRES to be FIRST
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#else /* else must be windows */
#include <windows.h>
#endif /* defined(UNIX) */
#include <string.h>
// Primary JADE includes
#include "jomtypes.h"
#include "jomparam.hpp"
#include "jomobj.hpp"
#include "jomobjin.hpp"
#include "jomobjct.hpp"
```

Figure 23

The include files (Figure 23) are very similar to the ones defined in Figure 18 with the addition of extra JADE headers that are required by the JADE object manager C and C++ APIs.

There are different two signatures for external methods: one for primitive methods and the other for class methods. Both return an integer value. A return value of zero indicates success and a non-zero value makes JADE object manager raise an exception based on the number returned.

The two types of JADE object manager specific parameters to the external methods are **DskParam** and **DskBuffer**. In simple terms, a **DskParam** is a union of the JADE supported data types and classes. A **DskParam** is the primary data type for communicating with the JADE object manager. Accessing the contents of the **DskParam** is achieved via the use of **paramGet** / **paramSet** methods defined in **jomparam.hpp**. A **DskBuffer** is a low-level structure for holding the actual data of the object. The layout of a **DskBuffer** is hidden to allow freedom of implementation.

```
// Ease of use macro
#if defined(UNICODE)
#define STRCMP(_a, _b)  wcsncmp( _a, _b )
#else
#define STRCMP(_a, _b)  strcmp( _a, _b )
#endif
```

Figure 24

The C macro (Figure 24) is defined for ease of use, to avoid putting `#if defined(UNICODE)` all through the code

```
// Signature; String::isEqual(s2 : String) : Boolean
extern "C" DllExport int JOMAPI
primitive_compare(
    UINT primitive_number,
    DskParam* primitiveValue,
    DskParam* pParams,
    DskParam* pReturn)
{
    int result;

    // Pointer to data inside DskParam
    Character * s1;
    // JADE length of String,
    // including possible imbedded NULLs
    Size s1Length;

    result = paramGetString(
        *primitiveValue, s1, &s1Length);
    CHECK_RESULT;
    // CHECK_RESULT is a shorthand way of writing
    // if (result != 0) return result;

    Character * s2;
    Size s2Length;
    result = paramGetString(
        *pParams, s2, &s2Length);
    CHECK_RESULT;

    // return the answer as a Boolean,
    // to the sender using pReturn
    Boolean answer;
    answer = STRCMP( s1, s2 ) == 0 ? TRUE : FALSE;

    result = paramSetBoolean(*pReturn, answer);
    CHECK_RESULT;

    // Returning 0 indicates everything "worked"
    // NON-Zero will raise a JADE exception
    return 0;
}
```

Figure 25

In the `primitive_compare` function (Figure 25), the receiver primitive is passed as the `primitiveValue` parameter. The `paramGetString` updates the Character pointer (`s1` or `s2`) and the respective `length` parameter. If this external primitive method needed to update the String primitive type, care should be taken not to write more than length characters back into the buffer. The last parameter, `pReturn`, is updated (using `paramSetBoolean`) with the results of the work performed by this external method. If you use the wrong `paramSet` method (for example `paramSetOid`, when the correct code would have been `paramSetBoolean`) then you are likely to get an *Illegal type conversion attempted* exception raised on return from your external method.

```
// Signature; Text::isEqual(s2 : String) : Boolean
extern "C" DllExport int JOMAPI
object_compare(
    DskBuffer* pBuffer,
    DskParam* pParams,
    DskParam* pReturn)
{
    int result;
```

```

// Declare the receiver object
// so we can use the jomobj C++ wrappers
DskObject self(&pBuffer->oid);

// Get the length of the String property (s1)
// from the current object
Size s1Length;
result = self.getProperty(
    TEXT("s1"), &s1Length, __LINE__);
CHECK_RESULT;

// Allocate a buffer to hold the data
Character * s1 = new Character[ s1Length + 1];
if (s1 == 0)
    return OUT_OF_MEMORY; // Exception number

// Get JOM to copy String data into new'ed buffer
result = self.getProperty(
    TEXT("s1"), s1, __LINE__);
OUT_ON_RESULT;
// OUT_ON_RESULT is a shorthand way of writing
// if (result != 0) goto out;

// Get pointer to String data, and JADE length
Character * s2;
Size s2Length;
result = paramGetString(
    *pParams, s2, &s2Length);
OUT_ON_RESULT;

// Do the real work
Boolean answer;
answer = STRCMP( s1, s2 ) == 0 ? TRUE : FALSE;

// Return the results
result = paramSetBoolean(*pReturn, answer);

out:
delete [] s1;
return result;
}

```

Figure 26

Note: The use of `__LINE__` in a number of JADE object manager APIs allows you to locate the offending line of code if an exception is raised. For example, Error Code 1011 (Requested property is not defined for this object class) caused by `object_compare` in `extMethods` at line number 120.

The `object_compare` function (Figure 26) illustrates a number of points. The `DskObject self(&pBuffer->oid);` line is the normal way of converting from a `DskBuffer` to a `DskObject`. The advantage of using `DskObjects` is that this allows you to use the C++ proxy methods, as in `self.getProperty(...)` (defined in `jomobjin.hpp`). The property name `s1` is enclosed in the `TEXT("...")` macro, this allows the same code to be compiled for both ANSI and Unicode execution. There are a pair of calls to `getProperty`: the first is to obtain the length of the string property allowing you to create a buffer large enough to hold the data, and the second to retrieve the character data. The second string `s2` you can access via `paramGetString`, as this is a primitive parameter that you do not intend to modify. When allocating memory, take care to release it before exiting from the function.

```

class CharacterBuffer
{
public:
    CharacterBuffer()        { pBuf = 0; }
}

```

```

~CharacterBuffer()      { delete [] pBuf; pBuf = 0; }
Character * allocate(Size len) {
    if (pBuf != 0) delete [] pBuf;
    pBuf = new Character[ len + 1 ];
    return pBuf;
};
operator Character* () { return pBuf; }
private:
    Character * pBuf;
};

```

Figure 27

The **CharacterBuffer** class (Figure 27) is a simple class that ensures that any memory it allocates is freed when the C++ object is destroyed. It is used as part of the example **object_property** external method.

```

// Signature; Text::isSame(o2 : Text) : Boolean
extern "C" DllExport int JOMAPI
object_property(
    DskBuffer* pBuffer,
    DskParam* pParams,
    DskParam* pReturn)
{
    int result;

    DskObject self(&pBuffer->oid);

    Size s1Length;
    result = self.getProperty(
        TEXT("s1"), &s1Length, __LINE__);
    CHECK_RESULT;

    CharacterBuffer s1;
    if (s1.allocate(s1Length) == 0)
        return OUT_OF_MEMORY;

    // Get JOM to copy data into allocated buffer
    result = self.getProperty(
        TEXT("s1"), s1, __LINE__);
    CHECK_RESULT;

    // Get the object (o2)
    // that was passed as a parameter
    DskObject o2;
    result = paramGetOid(*pParams, o2.oid);
    CHECK_RESULT;

    // Get the length of the String property (s1)
    // from parameter object (o2)
    Size s2Length;
    result = o2.getProperty(
        TEXT("s1"), &s2Length, __LINE__);
    CHECK_RESULT;

    CharacterBuffer s2;
    if (s2.allocate(s2Length) == 0)
        return OUT_OF_MEMORY;

    // Get JOM to copy data into allocated buffer
    result = o2.getProperty(
        TEXT("s1"), s2, __LINE__);
    CHECK_RESULT;

    // Do the real work
    Boolean answer;
    answer = STRCMP( s1, s2 ) == 0 ? TRUE : FALSE;
}

```

```
    // Return the results
    return paramSetBoolean(*pReturn, answer);
}
```

Figure 28

The **object_property** external method (Figure 28) performs very similar actions to the **object_compare** external method. The major difference is that it uses the **CharacterBuffer** class.

The **object_compare** external method works successfully if any of the JADE object manager methods called do not raise an exception. If a JADE exception is raised, it is implemented by throwing a C++ exception. If a C++ exception occurs then any resources allocated in a function need to be released. In the **object_property** (and **object_compare**) function the resource is dynamically allocated memory. The **CharacterBuffer** class handles releasing of memory if a C++ exception is thrown. The most likely case where an exception could occur is where the object parameter **o2** may be null and the **o2.getProperty()** method call could raise a JADE object manager 1090 *Attempted access via null object reference* exception.

Design and Coding Issues

As is often stated, good planning and design can save a lot of subsequent work later in the development cycle. This is also true when developing external interfaces from JADE. There are a number of key development choices to be made and the implications of the each decision should be clearly understood.

The following topics give a brief discussion and list a number of points that you should think about when writing external interfaces.

Operating System Portability: Windows and Unix

JADE is available on both Windows and Unix. Do you want your external interfaces to be available on both these platforms?

Sometimes the answer is simple if your interface requirements are restricted to the GUI and the end-user interface and you therefore only need to support Windows. But what about the times when the external interface has no end-user component? If native code was written because it is handling performance-critical work, it should be close to the data it uses. This may mean that it needs to work on multiple operating systems. If multiple operating systems are being supported, the following is a list of points that you should take into account when developing your native code.

- Different sets of native APIs
- UNIX file system names are multiple-byte, not wide character, Unicode
- Different directory and file naming conventions, \ versus ./, and no drive letters
- Case sensitivity of directory and file names
- Different end-of-line terminators for text files

There are also developer issues to also take into account. Being on a different operating system, you will have different tools and commands to become accustomed to.

- Different tools to view and monitor processes and files
- Different set of development tools, such as editors
- Differing levels of C and C++ compiler standards compliance
- Different debugger and support tools

Central Processing Unit: More than just Intel

Related to the operating system question is the issue of what CPU drives the operating system. The JADE release on AIX RS/6000 hardware is driven by the Power family of CPUs. These use a big-endian byte ordering scheme, whereas Microsoft Windows operating systems run on Intel CPUs, which use little-endian byte ordering.

JADE has hidden this difference from the JADE programmer (excluding the case of Binary primitive types). Writing your C or C++ code in a clean way often deals with most of the endian-related issues. For example, a problem occurs when passing the address of a byte to a function that expects the address of an integer. This may work on little-endian hardware but will almost certainly cause stack data corruptions on big-endian hardware.

- Write byte order neutral code
- Alignment and packing of C structures can differ

Multilanguage: ANSI versus Unicode

The JADE product is shipped as two sets of binaries and database files. These are the ANSI, which is a single-byte character set, and Unicode, which is a two-byte character set. The ANSI version of JADE is the most common, but there is a large number of people who require more than a single byte character set to represent their language.

If you decide to not limit your external interfaces to just ANSI-based version of JADE, your code needs to handle multiple character encoding schemes. In the supplied JADE header files there are C macros and typedefs with which the JADE product is built. These are available to developers of external interfaces.

- The UNICODE define is recommended to control build behavior
- Use the data type **Character** is the correct size
- Remember the size of **Character** may not be one byte
- The macro **TEXT**("string") converts literals to the correct type

- Byte ordering affects Unicode characters
- Compilers may have format strings for the **printf** subroutine when printing Unicode characters
- Two-byte characters are not the only representation of the Unicode character set

Environment: Differences between Machines

Unless you have total control over all of the components and environments you install your JADE solution on, there are differences from installation to installation. Even on the same operating system there can be differences from one environment to the next. This is both a system administration and a developer issue. Wherever possible, make your code flexible and handle unexpected situations gracefully.

- Check to see if components are installed before using them
- Soft-code command names and file locations wherever reasonable

Compatibility: Consistency with JADE Behavior

External functions and methods run inside the same operating system process as the rest of JADE. This means a problem in native code can have a significant impact on the JADE environment.

The quality of the external interface library and the JADE product combine in the eyes of the end-user. JADE object manager has certain expectations of the behavior of JADE and external methods and functions. These need to be followed to ensure the smooth operation of the JADE solution.

- Do not use C language assert unless you intend to take down the JADE node
- Communicate with JADE only on the thread on which your native code was called
- Make your code thread-safe
- Make your code re-entrant safe if you call other JADE methods

Caution: You may find function entry points in the AIX implementation that have the same name and signature as Microsoft Windows APIs but do not use these functions. They may not be complete implementations and may have unexpected side effects. They could disappear at any time the containing library is re-released.

Summary

Although the JADE language and environment provide you with a lot of features and functionality, sometimes your requirements go beyond what has been provided and this is when you may require external interfaces.

The advantage of writing external interfaces is that it gives you access to specialized or optimized functionality that JADE does not provide. There are the corresponding disadvantages in that you need to handle more technical and design issues as well as manage more components to your overall application solution.