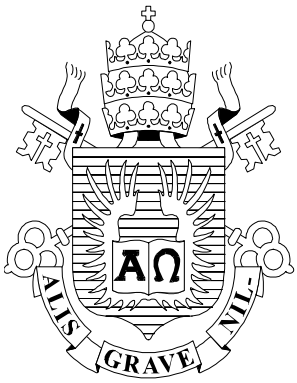


PUC

ISSN 0103-9741

Monografias em Ciência da Computação
03/02

Utilizando JNI para Adicionar Implementação Nativa C ou C++ a Programas JAVA

Felipe Carasso
Arndt von Staa

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900
RIO DE JANEIRO - BRASIL

PUC- RIO - DEPARTAMENTO DE INFORMÁTICA ISSN 0103-9741

Monografias em Ciência da Computação, Nº 03/02

Editor: Carlos J. P. Lucena Março, 2002

Utilizando JNI para Adicionar Implementação Nativa C ou C++ a Programas JAVA *

Felipe Carasso

Arndt von Staa

* Trabalho patrocinado pelo Ministério de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

Responsável por publicações:

Rosane Teles Lins Castilho
Assessora de Biblioteca, Documentação e Informação
PUC-Rio - Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453-900 - Rio de Janeiro, RJ, Brasil
Tel.: +55 21 3114-1516
Fax: +55 21 3114-1530
E-mail: bib-di@inf.puc-rio.br
Ftp-site: ftp.inf.puc-rio.br/pub/docs/techreports
Web-site: http://www.inf.puc-rio.br

Utilizando JNI para Adicionar Implementação Nativa C ou C++ a Programas JAVA

Felipe Carasso

Arndt von Staa

Departamento de Informática, PUC-Rio
Rua Marquês de São Vicente 225
22453-900 Rio de Janeiro, RJ, Brasil
Tel: +(55)(21) 3114-1500
hrimhari@rdc.puc-rio.br
arndt@inf.puc-rio.br

PUC-Rio.Inf.MCC03/02 Março, 2002

Abstract

The JAVA programming language has been widely adopted for its portability and user friendliness. However it is quite inefficient when dealing with compute bound operations. There are a couple of ways to get around this drawback, one of them deals with the utilization of native code. This report presents a detailed explanation of how to add native code to JAVA programs by using the Java Native Interface, or JNI. Most of the JNI specifications are examined in this report and examples were created and tested to illustrate them.

Keywords

Java, Java Native Interface, JNI, Java optimization, Portability.

Resumo

A linguagem de programação JAVA têm sido extensamente adotada devido às suas características de portatilidade e amicabilidade ao usuário. No entanto, a linguagem tem-se mostrado ineficiente em aplicações que consumam muitos recursos. Entre os modos de contornar este problema, existe um que envolve a utilização de código nativo em conjunção com código redigido em Java. Este relatório explica de forma detalhada como adicionar código nativo a programas JAVA através da utilização de Java Native Interface, ou JNI. É examinada a maior parte das especificações de JNI e foram criados e testados exemplos para ilustrá-las.

Palavras Chave

Java, *Java Native Interface*, JNI, Otimização de programas em Java, Portatilidade.

Sumário

1. Introdução

A linguagem de programação Java vem ganhando cada vez mais adeptos devido à sua simplicidade, riqueza de bibliotecas e portatilidade entre plataformas e sistemas operacionais.

No entanto, também existem fatores negativos tais como recursos previstos porém não implementados ("Este recurso será implementado em versão futura"), as reestruturações que sofre de versão para versão e o consumo de recursos computacionais caso o programa requeira um processamento um pouco mais intenso. Mesmo assim, é vantajoso utilizar esta linguagem em aplicações intensas em recursos[1], desenvolvendo-se programas híbridos compostos por código em Java e C ou C++.

A solução proposta pela Sun, empresa responsável pela criação e atualização da linguagem Java e do seu ambiente de execução (JVM – *Java Virtual Machine*), é a possibilidade de execução, de dentro do programa Java, de código nativo, compilado para uma determinada plataforma a partir de código C ou C++. Esta interface para a programação híbrida é chamada de *Java Native Interface*, ou JNI.

Ao usar a JNI – *Java Native Interface* – implementa-se em Java as partes do programa para as quais esta linguagem melhor se adeqüe, como por exemplo interfaces com o usuário e com a rede, deixando as partes intensivas em processamento ou recursos locais para serem implementadas em C ou C++ gerando código nativo. Tomando-se alguns cuidados com o projeto e a implementação da parte nativa é de se esperar que os programas resultantes sejam portáteis, ou seja, recompilando-se a parte nativa pode-se transferir o programa rapidamente para outras plataformas.

Como a JNI faz parte da especificação de Java, a maioria das *Java Virtual Machines* (JVM) implementam-na, assim como os *Java Development Kits* (JDK) provêm suporte apropriado. JNI não é suportado pelas JVM e pelos JDK oferecidos pela Microsoft até então, já que, como de costume, esta empresa decidiu elaborar suas próprias soluções.

1.1. Objetivos deste artigo

- Servir de referência para o programador que deseja desenvolver aplicações usando uma solução híbrida, utilizando a *Java Native Interface* - JNI, na qual parte do código é implementado em Java e a outra parte é implementada em C ou C++.
- Explicar, de forma suficientemente completa, a JNI, ilustrando as técnicas por intermédio de exemplos aplicáveis a futuros projetos.

1.2. Organização do artigo

No capítulo 2 apresentamos uma visão resumida da JNI. No capítulo 3 apresentamos as regras de interface da JNI. No capítulo 4 é ilustrado o uso da interface

[1] Uma **aplicação intensa em recursos** realiza demorados processamentos entre interações consecutivas com o usuário, ou realiza um volume grande de operações de leitura e gravação em arquivos, ou realiza um volume grande de construção ou destruição de objetos.

através de diversos exemplos. Finalmente no capítulo 5 damos uma retrospectiva geral sobre o uso de JNI.

2. Explicação resumida da JNI

A JNI fornece um conjunto de funções C ou C++ capazes de interagir com objetos Java. Estas funções são compostas por funções genéricas e outras específicas do programa que está sendo desenvolvido e que são utilizadas pelo código nativo, isto é C ou C++, para possibilitar a interação com o código escrito em Java. Os componentes nativos são compilados de modo que se crie uma biblioteca dinâmica (ex. .DLL, .so) que será importada pela classe Java que estabelece a interface com o código nativo. Desta forma o código escrito em Java poderá interagir com o código escrito em C ou C++.

Do lado Java, passam-se objetos como parâmetros para a função nativa que, por sua vez, pode alterar dados destes objetos, executar seus métodos e retornar valores que também podem ser objetos Java. Desta forma, é transparente para o programa que a função chamada é nativa e não Java.

Do lado C ou C++ deve-se implementar as funções a serem utilizadas em Java seguindo a convenção especificada por JNI para a escolha dos protótipos das funções (métodos). Os componentes C ou C++, uma vez em execução, podem utilizar quaisquer recursos, como se fossem um programa *stand-alone* redigido nestas linguagens. Isto significa que se aplicam todas as preocupações quanto ao uso de memória alocada dinamicamente, vazamento de memória, uso correto de ponteiros, perda de referência a recursos tais como arquivos abertos, etc.

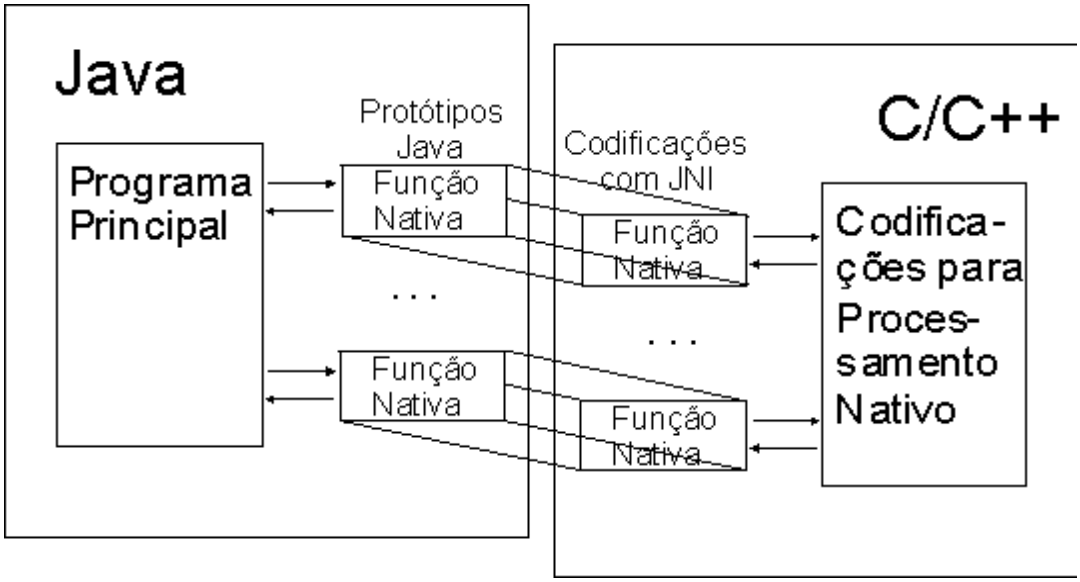
Um outro problema a ser observado é o da portatilidade destas funções. Além de uma biblioteca dinâmica criada em uma plataforma não servir em outra, muitas funções C ou C++ são dependentes do compilador, do ambiente de desenvolvimento e possivelmente até do sistema operacional utilizado ao compilar. É aconselhável, portanto, evitar o uso de recursos potencialmente incompatíveis entre plataformas de desenvolvimento e de execução, como, por exemplo, o uso de rotinas de interface com o usuário, uso de características complexas, uso de bibliotecas vinculadas a uma dada plataforma. Sem tomar cuidados de assegurar programas nativos portáteis, pode-se ter mais desvantagens do que vantagens ao utilizar JNI. O ideal a alcançar é a possibilidade de transportar componentes nativos entre plataformas requerendo meramente a recompilação e composição das bibliotecas dinâmicas utilizadas.

2.1. Organização aconselhável para programas utilizando a JNI

Considera-se um bom projeto JNI aquele que delimita bem as funções de interface entre as linguagens. Em particular, as funções em C ou C++ que têm contato direto com Java devem traduzir os dados passados como parâmetros e encaminhá-los às funções da implementação nativa. Estas funções realizarão efetivamente o trabalho que se espera da parte nativa. Devem existir também funções que traduzam de volta para o ambiente Java os resultados produzidos pelas funções nativas.

Desta forma fica perfeitamente identificado o que é interface entre os ambientes e o que é codificação de cada um. É claro que se tem um pequeno desperdício de esforço de processamento sempre que se cruza a interface. Este esforço será tão mais des-

prezível quanto maior for o esforço computacional médio consumido pelo lado nativo a cada cruzamento.



Arquitetura padrão proposta para a interface via `JNI`.

Outra vantagem desta arquitetura é a possibilidade de se substituir as funções nativas por funções em Java sem que se tenha que alterar a chamada ou a forma de se lidar com os parâmetros passados e recebidos. O mesmo ocorre com a parte em C ou C++, o que possibilita o teste independente das duas implementações. Estas propriedades da interface proposta permitem migrar-se classes de Java para o lado nativo sempre que for demonstrado que objetos destas classes são responsáveis por uma parte substancial do consumo de recursos computacionais.

2.2. O lado Java da JNI

Pouca coisa deve ser explicitada em Java ao redigir programas que utilizem a `JNI`. Os métodos que serão implementados em C ou C++ são declarados em qualquer classe como sendo `private native`, retornando valores de qualquer tipo, tais como `void`, `int`, `string`, ou mesmo objetos. Os exemplos estudados aqui utilizam diversos destes tipos de retorno.

Nas classes que contém chamadas para funções nativas, deve ser indicado o uso da correspondente biblioteca dinâmica que contém o código nativo. A forma para se indicar isso é utilizar o método `System.loadLibrary("nomeDaBiblioteca")`.

A biblioteca dinâmica é gerada a partir de código C ou C++ e a compilação varia de ambiente para ambiente. Em ambiente Windows Win32 deve ser gerada uma biblioteca `.DLL`. Já em ambiente Unix deve ser gerado um *shared object* `.so`.

É necessário tomar cuidado com o local onde se colocará a biblioteca dinâmica: em Win32 basta colocar a biblioteca no mesmo diretório em que se encontra a classe compilada, arquivo `.CLASS`. Em Unix a biblioteca deverá estar em um diretório indicado na variável `LD_LIBRARY_PATH`.

2.3. O lado C / C++ da JNI

O JDK da Sun oferece o aplicativo `javah` que visa facilitar a implementação do lado nativo dos métodos. Uma vez definidos na classe Java, pode-se utilizar este aplicativo para gerar um arquivo `.h` contendo os protótipos dos métodos da interface na forma C / C++. O módulo de definição contém as declarações C / C++ necessárias para se compilar corretamente os componentes dos módulos nativos que interagirão com a JNI. Não é necessário, no entanto, prender-se ao `javah` . A regra para criação dos protótipos é sempre a mesma, está bem definida e pode ser executada pelo próprio programador. Mesmo assim, recomenda-se fortemente o uso de `javah` pois reduz o número de possíveis erros de programação.

Tanto o arquivo gerado pelo `javah`, quanto o criado pelo programador, caso este opte pelo trabalho manual, devem incluir dois arquivos localizados no pacote do JDK. Estes arquivos de inclusão (#include) podem criar algumas dificuldades, pois mesmo que se utilize o `javah` para gerar o arquivo `.h`, estes arquivos de inclusão são adicionados como se fossem arquivos do compilador usando um comando `#include <nome-do-arquivo>`, sem qualquer indicação de onde estão os arquivos de inclusão. Para resolver este problema, o programador deve configurar corretamente o seu ambiente de desenvolvimento C ou C++ de modo que os arquivos de inclusão sejam encontrados durante a compilação. Em geral os compiladores permitem que se defina uma lista de diretórios (*path*) que contém o diretório em que se encontram os arquivos de inclusão. Caso isto não seja possível tem-se o recurso, pouco recomendado, de copiar os arquivos em questão para o diretório das inclusões do compilador. Não é recomendável pois estes arquivos podem perder-se ou tornar-se incompatíveis quando for atualizado o compilador ou a *Java Virtual Machine*

Se por um lado existe este pequeno inconveniente, um código fonte elaborado numa plataforma segundo este padrão possui maior compatibilidade com outra, pois o programador não terá que se prender ao local onde está instalado o JDK.

Para adquirir, atribuir e devolver valores para um objeto Java, é necessário usar o pacote de funções padrão C ou C++ que lidam com os elementos Java passados para a implementação. Parâmetros primitivos (`int`, `float`, etc.) podem ser utilizados sem necessidade de qualquer tradução e podem ser retornados da mesma forma. No caso de *strings*, no entanto, é prudente lembrar-se de liberar a memória ocupada pelo *string* antes de retornar.

2.4. Gerar o `.h` com Javah

Se o programador optar por utilizar o `javah` para gerar o `.h` com as definições das funções nativas correspondentes aos métodos definidos na classe Java, o programador deve primeiro compilar a classe:

```
javac arquivo.java
```

Serão gerados diversos arquivos `.class`, um para cada classe contida no arquivo `.java` compilado. Para gerar o `.h`:

```
javah –jni Nome-Da-Classe
```

Será gerado o `.h` e que obedece ao formato descrito na seção a seguir.

2.5. Estrutura do .h gerado pelo Javah

```
 1 #include <jni.h>

 2 #ifndef _Included_nomeClasse
 3 #define _Included_nomeClasse
 4 #ifdef __cplusplus
 5 extern "C" {
 6 #endif
 7 /*
    * Comentários sobre o método nativo aqui...
    */
 8 JNIEXPORT tipoRetorno JNICALL Java_nomeClasse_nomeMétodo
 9   (JNIEnv *, jobject[, tipo-parâmetro[…]]);

10 ...

11 #ifdef __cplusplus
12 }
13 #endif
14 #endif
```

A seguir explicamos cada uma das linhas que se encontram numeradas

1. O arquivo `.h` inclui o arquivo `jni.h` do sistema, localizado no diretório `include` do pacote `JDK`. Este arquivo, por sua vez, inclui o arquivo `jni_md.h`, que está no diretório "`include/ambiente`", onde "`ambiente`" é o nome do ambiente utilizado. Em Win32, o nome de "`ambiente`" é `win32`. Em Linux, o nome de "`ambiente`" é `genunix`. Assumindo-se que o pacote `JDK` esteja instalado em "`/usr/jdk`", o caminho até o arquivo `jni.h` é "`/usr/jdk/include/jni.h`", e o diretório a ser incluído na compilação da biblioteca é "`/usr/jdk/include`". De modo análogo, se o ambiente é Linux, o caminho até o arquivo `jni_md.h` é "`/usr/jdk/include/genunix/jni_md.h`" e o diretório a ser incluído é "`/usr/jdk/include/genunix`".
2. Para evitar que o arquivo de inclusão seja compilado mais de uma vez na mesma compilação, o precompilador verifica esta definição. O que vem depois só é interpretado se esta definição ainda não existe.
3. Esta definição impede que o arquivo seja interpretado caso seja referenciado mais uma vez no código a ser compilado.
4. Verifica se está sendo usado C ou C++. Trabalha em conjunto com (5), (6), (11), (12) e (13). Esta estrutura torna o arquivo compatível com ambas as linguagens.
5. Insere os dados deste arquivo no modo de compatibilidade com C do C++.
6. Termina o controle de dependência da linguagem C ou C++.
7. `Javah` insere comentários sobre cada método nativo criado, logo antes de seus protótipos. Estes comentários contém o nome da classe, o nome do método em Java e as classes dos objetos usados como parâmetros ao método. Veja `jniref.html` (ou `http://java.sun.com/docs/books/tutorial/native1.1/summary`), *Encoding for Java Type Signatures* para melhor compreender os diversos tipos de parâmetros na forma de assinatura.

8. Definição do protótipo da função nativa que implementa o método. Sempre começa com `JNIEXPORT`, seguido do tipo de retorno do método, seguido de `JNICALL`, e finalmente o nome da função, composto pela concatenação: `Java_` + <*nome da classe que contém o método*>`_` + <*nome do método*>. O tipo de retorno é um dos fornecidos por `JNI`: `jboolean`, `jbyte`, `jchar`, `jshort`, `jint`, `jlong`, `jfloat`, `jdouble` ou `void`. Para mais detalhes consulte: `jniref.html` *(*ou `http://java.sun.com/docs/books/tutorial/native1.1/summary`*)*, *Mapping Between JNI and Native Types* para melhor compreender os diversos tipos de retorno. No caso dos nomes das classes e/ou dos métodos devem ser utilizados caracteres *Unicode*, a conversão é feita baseada na seguinte regra: "_0XXXX" corresponde ao caracter Unicode XXXX. "_1" corresponde ao caracter "_". "_2" corresponde ao caracter ";". "_3" corresponde ao caracter "[".
9. Parâmetros passados à função. `JNIEnv` que constitui um portal de comunicação entre o lado nativo e o lado Java. O primeiro `jobject` também sempre é passado: ele é um ponteiro para o objeto que chamou este método nativo. Depois há um tipo para cada parâmetro definido para o método. Veja `jniref.html` (ou `http://java.sun.com/docs/books/tutorial/native1.1/summary`), *Mapping Between JNI and Native Types* para melhor compreender os diversos tipos de parâmetros.
10. Para cada método definido, repetem-se os itens de (7) a (9).
11. Verifica se a linguagem é C++, para...
12. ... encerrar a estrutura de compatibilidade com a linguagem nativa usada.
13. Fim da verificação da linguagem em uso (C ou C++)
14. Fim da verificação de múltiplas inclusões do arquivo, iniciada em (2).

3. Aprofundando-se na JNI pelo lado C ou C++

Neste capítulo descreveremos as regras a serem observadas do lado do código nativo, ao desenvolver programas que interagem através da JNI. Para uniformizar o texto adotamos a forma C de interface, alertando que a forma C++ pode ser obtida conforme descrito a seguir.

3.1. Interação com JNI: propriedades específicas a C e C++

C e C++ diferenciam-se ligeiramente na forma como interagem com `JNI`. Existem duas diferenças na forma de se utilizar as funções fornecidas pelo pacote `JNI`. Estas diferenças dizem respeito ao parâmetro `JNIEnv* pEnv` passado para a implementação nativa. Este parâmetro é, na realidade, um ponteiro para uma estrutura de dados e de ponteiros para funções típica de um objeto C++. Em C utiliza-se esta estrutura na forma convencional como um ponteiro para um espaço de dados. Em C++ utiliza-se `pEnv` diretamente como um ponteiro para um objeto.

Detalhando, em C, ao utilizar uma função do pacote `JNI`, é necessário dereferenciar[2] `pEnv` antes de referenciar a função a ser executada. Além disso o primeiro pa-

[2] *Dereferenciar* é a operação de transformar um ponteiro no valor por ele apontado, tipicamente pelo caminhamento do ponteiro para o espaço de dados por ele apontado.

râmetro da função deve sempre ser `pEnv`. Ou seja, em C uma chamada a uma função `JNI` tem o formato:

```
( *pEnv )->função( pEnv, outros parâmetros );
```

Já em C++, o parâmetro não precisa ser dereferenciado pois é tratado como um ponteiro para um objeto. Por este mesmo motivo, ele também não precisa ser passado à função, uma vez que esta recebe automaticamente o ponteiro. Uma chamada a uma função `JNI` em C++ tem consequentemente o formato:

```
pEnv->função( outros parâmetros );
```

Mais tarde será detalhado como redigir o protótipo das funções, indicando o tipo retornado, o nome e os parâmetros esperados. No entanto, o programador não deve esquecer de utilizá-las sempre através de uma chamada indireta através do ponteiro `JNIEnv * pEnv`.

3.2. Como obter valores de Java no código nativo

Somente objetos visíveis nos parâmetros passados às funções de interface da implementação nativa podem ser acessados por estas funções. O primeiro parâmetro – desconsiderando o ponteiro `JNIEnv *` - é passado por JNI independentemente da definição do protótipo e equivale ao objeto que chamou a função nativa, fornecendo acesso a tudo o que este objeto permite ser visto. Superficialmente tratam-se de elementos do próprio objeto ou métodos contidos nele. Outros objetos podem ser passados como parâmetro à função nativa, obedecendo aos mesmos critérios.

Se um parâmetro primitivo for passado (`int`, por exemplo), este parâmetro será acessado diretamente sem ser necessária a utilização de uma função JNI para traduzi-lo. O programador deve ter apenas o cuidado de saber lidar com o tamanho em bits deste tipo primitivo e utilizar os tipos nativos certos. Veja *jniref.html* (ou *http://java.sun.com/docs/books/tutorial/native1.1/summary*), *Mapping Between JNI and Native Types*. Lá é feita uma relação direta entre os tipos primitivos do lado Java e os tipos definidos por JNI como equivalentes aos primitivos Java para o lado C ou C++, incluindo o tamanho em bits.

Se o parâmetro passado for um objeto, então qualquer operação que o envolva deverá ser feita por via de uma função JNI. Desta forma, o ambiente Java permanece razoavelmente encapsulado e sua estrutura interna é irrelevante para a implementação nativa.

Um elemento de um objeto não é acessado diretamente. Ao invés disso, a VM cria uma cópia para o usuário.

Antes de acessar um elemento, o programador deve determinar a que classe o objeto pertence e, depois, adquirir um identificador referenciando este elemento. Então o acesso a elementos de um objeto pode ser separado em duas etapas: adquirir o identificador deste elemento (determinando a classe) e adquirir o valor através deste identificador.

Para adquirir a classe a que o objeto pertence, utiliza-se a seguinte função (notação C):

```
jclass GetObjectClass( JNIEnv* pEnv, jobject obj );
```

na qual pEnv é o ponteiro do tipo JNIEnv * recebido pela função nativa e obj é o objeto do qual se deseja determinar a classe.

Para interagir com um atributo da classe é necessário primeiro obter o seu identificador. Com este identificador pode-se então interagir com o atributo. Para adquirir um identificador, quando se conhece a classe do objeto, utiliza-se a seguinte função se o elemento não for estático:

```
jfieldID GetFieldID( JNIEnv* pEnv, jclass classe,
const char* pNome, const char* pTipo );
```

Se o atributo for estático a função é:

```
jfieldID GetStaticFieldID( JNIEnv* pEnv, jclass
classe, const char* pNome, const char* pTipo );
```

onde pEnv é o ponteiro do tipo JNIEnv * adquirido pela função nativa, *classe* é a classe obtida com a função anterior, pNome é um *string* contendo o nome do elemento a ser identificado e pTipo é um *string* que identifica o tipo deste elemento, de acordo com a tabela apresentada em jniref.html (ou http://java.sun.com/docs/books/tutorial/native1.1/summary), *Encoding for Java Type Signatures*. Estas funções podem disparar as exceções NoSuchFieldError, para o caso do elemento não poder ser encontrado, ExceptionInInitializerError, para o caso do elemento ser de uma classe não inicializada e JNI falhar ao inicializar a classe, e OutOfMemoryError, se o sistema não tiver memória suficiente.

Uma vez que o identificador seja conhecido, pode-se adquirir o valor deste elemento. Se o elemento não for estático, utiliza-se a função:

```
tipoNativo Get<i>Tipo</i>Field( JNIEnv* pEnv, jobject obj,
jfieldID id );
```

E se o elemento for estático:

```
tipoNativo GetStatic<i>Tipo</i>Field( JNIEnv* pEnv, job-
ject obj, jfieldID id );
```

onde pEnv e obj foram adquiridos pela função nativa, e id foi adquirido pela função anterior. tipoNativo é um dos diversos tipos primitivos de C ou C++ (int, por exemplo). tipo é o tipo primitivo do lado Java. Existem funções Get*tipo*Field ou GetStatic*tipo*Field para cada tipo conhecido por Java. O tema *Tipo* no nome das funções identifica o tipo do campo a ser acessado.

É possível guardar em variáveis estáticas os identificadores adquiridos numa primeira chamada à implementação nativa, para evitar ter que fazer estas tradução todas as vezes. Porém deve-se ter *extremo cuidado* para não excluir (delete) um objeto no meio do caminho entre a aquisição do identificador de um de seus elementos e o uso deste identificador.

3.3. Como atribuir valores do código nativo a Java

Não se pode atribuir valores primitivos diretamente a variáveis Java. No entanto, pode-se atribuir valores a variáveis membro de objetos. Neste caso, o caminho para atribuir valores é similar ao de adquirir valores. Deve-se primeiro adquirir o identificador do elemento do objeto a ter seu valor modificado, conforme descrito no tópico anterior, e depois utilizar funções JNI que atribuem valores a elementos.

Uma vez tendo posse do identificador, utiliza-se as seguintes funções para escrever o valor desejado no elemento Java:

```
void Set<i>Tipo</i>Field( JNIEnv* pEnv, jobject obj,
jfieldID id, tipoNativo valor );
```

e

```
void SetStatic<i>Tipo</i>Field( JNIEnv* pEnv, jobject obj,
jfieldID id, tipoNativo valor );
```

3.4. Como lidar com strings

Uma *string* não é considerada um tipo primitivo. Além disso, Java usa *strings* no formato Unicode. Ao adquirir valores de *strings* passadas por Java, a implementação nativa pode adquirir cópias traduzidas para ASCII ou cópias em Unicode mesmo.

Uma *string* Java pode ser adquirida como parâmetro para a função nativa ou como parte de um objeto passado como parâmetro. Para o caso da *string* ser parte de um objeto, ela deve antes ser identificada como descrito nos tópicos anteriores. Eventualmente o programador poderá sobrescrever uma *string* que é parte de um objeto como feito com outros tipos primitivos, conforme descrito anteriormente também.

`JNI` fornece funções capazes de lidar com ambos os casos ASCII e Unicode. Veja também os tópicos anteriores sobre como lidar com valores Java para maiores detalhes quanto a alguns dos parâmetros destas funções.

Para saber o comprimento de uma *string* Java:

```
jsize GetStringLength( JNIEnv* pEnv, jstring
string);
```

Esta função retorna o número de caracteres Unicode de *string*. Se o programador deseja o número de caracteres UTF-8 (compatível com ASCII), deve utilizar a função:

```
jsize GetStringUTFLength( JNIEnv* pEnv, jstring
string);
```

Para adquirir uma cópia de uma *string* Java, em Unicode:

```
const jchar* GetStringChars( JNIEnv* pEnv, jstring
string, jboolean* pIsCopy );
```

e para adquirir uma cópia no formato UTF-8:

```
const char* GetStringUTFChars( JNIEnv* pEnv,
jstring string, jboolean* pIsCopy );
```

Nestes dois casos o último parâmetro é um ponteiro para uma variável C/C++ que indicará ao programador se uma cópia da *string* Java foi feita ou não, através dos valores `JNI_TRUE` e `JNI_FALSE`. Não está muito claro nas especificações de JNI quando um ou outro caso ocorrerá, e provavelmente o programador jamais necessitará da informação devolvida em `pIsCopy` se sempre utilizar corretamente a *string* adquirida. Assim sendo, pode-se passar um ponteiro nulo no lugar de `pIsCopy` - neste caso a função não indicará se houve cópia ou não – e simplesmente verificar se a função retornou um ponteiro válido ou nulo (será nulo se houve falha ao devolver uma cópia da *string*).

Eventualmente o programador terá que explicitamente liberar a cópia adquirida. Para tanto deverá utilizar uma das funções seguintes de acordo com o tipo de cópia:

```
void ReleaseStringChars( JNIEnv* pEnv, jstring
string, const jchar* pUniStr );
```

ou

```
void ReleaseStringUTFChars( JNIEnv* pEnv, jstring
string, const char* pStr );
```

Para o caso de a implementação nativa precisar retornar uma *string* Java criada baseada em *strings* nativas, a JNI provê funções para criação de *strings* Java pelo lado nativo baseando-se em variáveis *string* nativas:

```
jstring NewString( JNIEnv* pEnv, const jchar* pUniStr, jsize
comprimento );

jstring NewStringUTF( JNIEnv* pEnv, const char* pStr );
```

Uma *string* Java criada a partir de Unicode precisa do comprimento explicitamente enunciado pela variável *comprimento*.

Ambas funções retornam uma *string* Java (`java.lang.String`) capaz de ser retornada pela função nativa e podem disparar a exceção `OutOfMemoryError` para o caso de não haver memória suficiente.

3.5. Como lidar com Vetores

A JNI fornece uma forma bastante interessante para o programador lidar com vetores. O conjunto de funções JNI para acesso a vetores pode ser dividido em três grupos: acesso completo a vetores de tipo primitivo, acesso parcial a vetores de tipo primitivo e acesso a elementos de vetores de objetos.

Se o vetor é de elementos de tipos primitivos (`int`, por exemplo), o programador pode optar por adquirir uma cópia completa do vetor ou uma cópia parcial. A diferença entre usar uma ou outra diz respeito ao desempenho do programa. Se o vetor for potencialmente grande e a implementação nativa estiver interessada apenas em alguns elementos, é mais interessante adquirir uma cópia parcial do que uma cópia do vetor inteiro.

As funções de acesso a vetores de tipo primitivo são:

```
tipoNativo* Get<Tipo>ArrayElements( JNIEnv* pEnv,
<Tipo>Array vetor, jboolean* pIsCopy );
```

onde `tipoNativo` e *Tipo* são equivalentes ao tipo primitivo dos elementos do vetor para o lado nativo e para o lado Java. Para o caso da definição do vetor, o tipo primitivo é seguido de *Array* (como em *jintArray*, por exemplo). Veja [3] para mais detalhes sobre os tipos primitivos, e a seção *Como lidar com strings* para maiores detalhes sobre o argumento `pIsCopy`. Esta função pode retornar um ponteiro nulo caso a operação não possa ser concluída.

```
void Get<Tipo>ArrayRegion( JNIEnv* pEnv, tipoArray
vetor, jsize inicio, jsize comprimento, tipoNativo*
pBuf );
```

onde `inicio` é o índice do primeiro elemento da área a ser copiada, `comprimento` o número de elementos a serem copiados a partir do elemento indicado por `inicio`, *Tipo* e `tipoNativo` são tipos primitivos para o lado nativo e o lado Java, equivalentes ao tipo primitivo dos elementos do vetor, e `pBuf` é uma área previamente alocada pelo programador para onde serão copiados os elementos. Esta área pode ser alocada conforme o gosto do programador, e deve ter como comprimento no mínimo o número de elementos a serem copiados vezes o tamanho em bytes do tipo de cada elemento (`comprimento*sizeof( tipo )`). Esta função pode disparar a exceção `ArrayIndexOutOfBoundsException`, para o caso do índice ou do comprimento indicado não ser válido.

Para o caso de se adquirir uma cópia completa do vetor, eventualmente esta cópia terá que ser liberada a exemplo das cópias de *strings*. As modificações no vetor copiado só terão garantia de serem espelhadas para o vetor Java ao executar esta função:

```
void Release<i>Tipo</i>ArrayElements( JNIEnv* pEnv,
tipoArray vetor, tipoNativo* copia, jint opcao );
```

onde *Tipo* e `tipoNativo` são os tipos primitivos conforme explicado anteriormente, `copia` é a cópia do vetor adquirida com a função descrita no início deste tópico e `opção` é uma entre as seguintes opções:

`0` (zero):	copia o conteúdo de `copia` e libera o espaço alocado para `copia`.
`JNI_COMMIT`:	copia o conteúdo de `copia` mas não libera o espaço ocupado.
`JNI_ABORT`:	libera o espaço ocupado por `copia` sem copiar o conteúdo de volta.

O mais comum é utilizar sempre a opção 0 (zero) para esta função, porém as opções `JNI_COMMIT` e `JNI_ABORT` podem ser úteis quando uma alteração em vetor for separada em etapas, cada uma com verificação de ocorrências indesejadas, de modo que uma etapa bem sucedida é seguida de *commit* e uma etapa mal sucedida acarreta em *abort*, a exemplo do que é feito em transações de banco de dados.

Para o caso de ter sido adquirida uma região do vetor apenas, modificações nesta região são garantidas de serem espelhadas para o vetor Java ao utilizar a função:

```
void Set<i>Tipo</i>ArrayRegion( JNIEnv* pEnv, tipoArray
vetor, jsize inicio, jsize comprimento, tipoNativo*
pBuf );
```

Esta função é análoga a função `Get`*`Tipo`*`ArrayElements` vista anteriormente no que diz respeito aos parâmetros e exceções, porém a área ocupada por `pBuf` deve ser liberada pelo próprio programador.

Para lidar com vetores de objetos, o programador precisará adquirir cada elemento por vez, ao invés de adquirir um pedaço ou todo o vetor de uma só vez. As funções de acesso a vetores de objetos são:

```
jobject GetObjectArrayElement( JNIEnv* pEnv, job-
jectArray vetor, jsize indice );

void SetObjectArrayElement( JNIEnv* pEnv, jobjec-
tArray vetor, jsize indice, jobject valor );
```

onde `indice` é o índice do elemento a ser manipulado e `valor` o valor a ser atribuído a ele.

As exceções que podem ser disparadas por estas funções são `ArrayIndexOutOfBoundsException`, para o caso do índice indicado não ser válido, e `ArrayStoreException`, para o caso da classe do valor a ser atribuído para um elemento não ser uma subclasse da classe do elemento.

É importante saber o comprimento total do vetor em diversas situações. A mais comum é a de quando o vetor todo for ser varrido por um `for( )`, onde necessita-se saber qual é o maior índice possível. Para adquirir o número de elementos de um vetor Java, utiliza-se a função:

```
jsize GetArrayLength( JNIEnv* pEnv, jarray vetor );
```

Esta função serve para qualquer vetor Java, seja ele primitivo ou de objetos.

Para o caso em que deseja-se devolver um novo vetor para o lado Java, é possível criar um vetor Java pelo lado nativo utilizando uma das funções:

```
tipoArray New<i>Tipo</i>Array( JNIEnvv* pEnv, jsize
tamanho );

jarray NewObjectArray( JNIEnv* pEnv, jsize tamanho,
jclass classeElementos, jobject valorInicial );
```

onde `tamanho` é o número de elementos do vetor, `classeElementos` é a classe a que os elementos pertencem e `valorInicial` é o valor inicial dos elementos até que sejam alterados pelo programa.

As duas funções podem disparar a exceção `OutOfMemoryError`, para o caso de não haver memória suficiente para o novo vetor ser criado.

3.6. Como executar métodos Java a partir do código nativo

É possível executar métodos Java pelo lado nativo, desde que o método a ser executado seja visível através dos objetos passados como parâmetros para a implementação nativa.

Para executar um método Java, é necessário adquirir um identificador para este método, a exemplo de como foi feito com elementos de um objeto na seção *Como obter valores de Java na implementação nativa.* A função que retorna este identificador é:

```
jmethodID GetMethodID( JNIEnv* pEnv, jclass classe,
const char* pNome, const char* pSig );
```

ou, se o método for estático da classe:

```
jmethodID GetStaticMethodID( JNIEnv* pEnv, jclass
classe, const char* pNome, const char* pSig );
```

Estas funções causam uma classe ainda não inicializada a ser inicializada e funcionam exatamente como as funções `GetFieldID( )` e `GetStaticFieldID( )` vistas no tópico “Adquirindo Valores de Java na Implementação Nativa”. Só que neste caso, a assinatura do método é um pouco mais complexa que a de um elemento. Métodos possuem um tipo de retorno e diversos tipos de parâmetros, e todos estes tipos são indicados na assinatura, conforme indicado em *jniref.html (*ou [3] seção *Encoding for Java Type Signatures*). Os tipos dos parâmetros são indicados entre parênteses, seqüen-

cialmente, e o tipo de retorno do método é indicado após o fecha parênteses. Por exemplo, se o método em questão retornasse uma *string* e tivesse como parâmetros um objeto e uma *string*, a assinatura seria "`(LclasseDoObjeto; Ljava/lang/String;)Ljava/lang/String;`". Note que a assinatura "`L`" é a única que necessita de ponto e vírgula. Outras assinaturas (`int` -- `I` --, por exemplo) são indicadas uma após a outra sem qualquer separador. Um método que não retorne valor (`void`) e tenha como parâmetros dois inteiros e uma *string* teria como assinatura "`(IILjava/lang/String;)V`".

Esta função pode disparar a exceção `NoSuchMethodError` caso não seja possível identificar o método pelo nome ou assinatura indicada, `ExceptionInInitializerError` caso a classe não tenha sido inicializada antes e não tenha sido possível inicializá-la e `OutOfMemoryError` se não houver memória suficiente.

Uma vez de posse do identificador do método, o programador pode executá-lo de diversos modos. Indicando os parâmetros por extenso, indicando os parâmetros através de um vetor de tipo `jvalue`, ou indicando os parâmetros através de uma `va_list` (ver `varargs` em C ou C++). Além disso, o programador pode desejar executar o método relacionado ao objeto (virtual), à classe a que o objeto pertence (estático) ou à superclasse relacionada ao objeto (não virtual). As funções para chamada de métodos relacionados ao objeto são:

```
tipoRetorno Call<Tipo>Method( JNIEnv* pEnv,
jobject objeto, jmethodID id, ... );

tipoRetorno Call<Tipo>MethodA( JNIEnv* pEnv, jobject
objeto, jmethodID id, jvalue* vtArgs );

tipoRetorno Call<Tipo>MethodV( JNIEnv* pEnv, jobject
objeto, jmethodID id, va_list args );
```

Estas funções executam um método relacionado a um objeto `objeto`, de tipo *Tipo*, que retorna `tipoRetorno`, indicando os parâmetros *por extenso*, através de um vetor `jvalue` e através de uma `va_list`, respectivamente.

Para executar o método relacionado à classe, utilize as funções análogas:

```
tipoRetorno CallStatic<Tipo>Method( JNIEnv* pEnv,
jclass classe, jmethodID id, ... );

tipoRetorno CallStatic<Tipo>MethodA( JNIEnv* pEnv,
jclass classe, jmethodID id, jvalue* vtArgs );

tipoRetorno CallStatic<Tipo>MethodV( JNIEnv* pEnv,
jclass classe, jmethodID id, va_list args );
```

neste caso o identificador *id* deve ter sido adquirido com a função *GetStaticMethodID( )*.

Para executar o método relacionado à superclasse:

```
tipoRetorno CallNonvirtual<Tipo>Method( JNIEnv* pEnv,
jobject objeto, jclass classe, jmethodID id, ... );

tipoRetorno CallNonvirtual<Tipo>MethodA( JNIEnv*
pEnv, jobject objeto, jclass classe, jmethodID id,
jvalue* vtArgs );
```

```
tipoRetorno CallNonvirtualTipoMethodV( JNIEnv*
pEnv, jobject objeto, jclass classe, jmethodID id,
va_list args );
```

Estas funções de execução de métodos podem disparar exceções relacionadas aos métodos executados.

3.7. Como obter assinaturas de métodos usando Javap

O pacote JDK da Sun oferece a ferramenta javap para auxiliar o programador na identificação de assinaturas e assim evitar erros ao tentar determinar as assinaturas por esforço próprio. Esta ferramenta é utilizada da seguinte forma.

Se o programador deseja identificar as assinaturas dos métodos da classe *MeuJNI*, contida no arquivo *MeuJNI.class*, deve executar:

```
javap –p –s MeuJNI
```

Esta ferramenta fará um *disassemble* na classe e listará os métodos identificando suas assinaturas no formato a ser usado em JNI.

Se por um lado isto torna a identificação das assinaturas mais fácil, é provável que o programador execute seu cronograma de projeto de uma forma que não seja favorável o uso desta ferramenta, assim como pode acontecer com o uso da ferramenta `javah` para gerar o `.h` a ser usado pela implementação nativa.

3.8. Como gerar bibliotecas dinâmicas em Win32

O formato de biblioteca esperado pela JVM quando utilizando JNI em ambiente Win32 é o de DLL. A DLL gerada deve ser depositada no mesmo diretório da classe que a utiliza e o nome dado ao arquivo DLL (sem a extensão .dll) é exatamente o nome a ser chamado no método *System.loadLibrary()*. Então se o programador gerar uma biblioteca de nome *MeuJni.dll*, a classe Java que for utilizar essa biblioteca deverá ter a seguinte chamada:

```
static
{
  System.loadLibrary( “MeuJni” );
}
```

Este método pode ativar duas exceções Java: `SecurityException` e `UnsatisfiedLinkError`. É provável que a segunda cause mais problemas, como por exemplo se o programador colocar a biblioteca no diretório errado ou importar a biblioteca com um nome diferente do devido.

Para gerar uma biblioteca utilizando VisualC++, o programador deve seguir os seguintes passos:

1. Criar um Novo Projeto escolhendo como tipo *Win32 Dynamic Library*.
2. Configurar os diretórios de include, adicionando os do JDK, utilizando a sequência de menus *Tools -> Options -> Directories* ou *Project -> Settings -> C/C++*, *Category: Preprocessor*, *Aditional Include Directories*. Se o JDK estiver instalado no diretó-

rio `c:\jdk13`, os diretórios a serem adicionados são `c:\jdk13\include` e `c:\jdk13\include\win32`.

3. Configurar o projeto para salvar a DLL gerada no mesmo diretório da classe que a importará utilizando a seqüência de menus *Project -> Settings -> Link -> Output File Name*.

Atenção! TurboC++ não sabe lidar com os formatos utilizados por JNI. Consequentemente não é possível gerar bibliotecas JNI por ele.

3.9. Como gerar bibliotecas dinâmicas em Unix

O formato de biblioteca esperado pela JVM no ambiente Unix é o de *Shared Object*, ou .so. Estes arquivos possuem extensão `.so` e um prefixo `lib` no nome do arquivo. Assim sendo, se foi escolhido o nome de `MeuJni` para a chamada do método `System.loadLibrary()`, a biblioteca Unix tem que ser gerada com o nome `libMeuJni.so`, lembrando que os sistemas Unix consideram diferentes letras maiúsculas de letras minúsculas.

No ambiente Unix, a biblioteca gerada pode estar, a princípio, em qualquer diretório. Porém deve ser configurada a variável de ambiente `LD_LIBRARY_PATH`, que contém a lista de diretórios com bibliotecas a serem usadas pelo usuário, para que aponte também para o diretório escolhido. Caso isto não seja feito é provável que seja gerada a exceção `UnsatisfiedLinkError`. A variável contém uma *string* identificando todos os diretórios com bibliotecas do sistema, portanto ela jamais deve ser sobrescrita e sim modificada para que contenha os diretórios que já continha adicionando-se o diretório onde o programador pretende depositar a biblioteca. É aconselhável que o diretório escolhido seja o mesmo da classe que importará a biblioteca, para que haja um padrão único tanto em Unix quanto em Win32.

Os diretórios de include do JDK são indicados no momento da compilação com a opção `-I`. Se, por exemplo, o JDK estiver instalado em `/usr/local/jdk123` devem ser adicionadas duas opções ao compilador:

```
-I/usr/local/jdk123/include
-I/usr/local/jdk123/include/genunix
```

(ou `-I/usr/local/jdk123/include/solaris` no caso do sistema operacional seja *Solaris*).

Para gerar uma biblioteca utilizando o compilador `gcc`, utilize a opção `-shared` para indicar que o resultado é um *shared object*, inclua a opção `-o` para salvar o arquivo no diretório correto e por fim indique os arquivos objeto ou os códigos fonte que devem gerar a biblioteca. Se o programador tiver feito um arquivo *MeuJni.c* com as implementações necessárias à biblioteca e for salvá-la no mesmo lugar com o nome de *libMeuJni.so* e o JDK estiver instalado como indicado acima, a linha de comando a ser usada é:

```
gcc -shared -I/usr/local/jdk123/include
-I/usr/local/jdk123/include/genunix -o libMeuJni.so
MeuJni.c
```

Se a implementação nativa for composta de vários arquivos .c, cada um deles deve ser primeiramente convertido em arquivo objeto e então devem ser utilizados os arquivos objeto para gerar a biblioteca. Este tipo de prática é bastante facilitada com o uso da ferramenta `make`. Se o programador tiver separado a implementação nativa nos

arquivos `jni-1.c` e `jni-2.c`, mantendo as condições restantes dos exemplos anteriores, a seqüência de compilação passa a ser:

```
gcc -c -I/usr/local/jdk123/include
     -I/usr/local/jdk123/include/genunix jni-1.c

gcc -c -I/usr/local/jdk123/include
     -I/usr/local/jdk123/include/genunix jni-2.c

gcc -shared -o libMeuJni.so jni-1.o jni-2.o
```

As duas primeiras linhas geram os arquivos `jni-1.o` e `jni-2.o`. Note que no momento de gerar a biblioteca não é mais necessário indicar os diretórios de `include`, porque os arquivos objeto já incluíram tudo o que precisavam.

4. Exemplos

Neste capítulo apresentaremos diversos exemplos de uso da JNI. Todos os exemplos foram desenvolvidos e testados usando-se:

- Windows NT e Windows 2000, `JDK 1.3` e o compilador MS-Visual C / C++.
- Linux 2.2.14-5.0 (kernel), `JDK 1.3` e o compilador GCC 2.91.66 (egcs).

4.1. Exemplo 1: Tipos primitivos como parâmetros

Este exemplo ilustra o tratamento de tipos primitivos passados por Java à implementação nativa.

Aqui a classe Java importa uma implementação nativa que tem como único objetivo receber valores de tipos primitivos e imprimi-los na tela. A função nativa pode receber infinitos parâmetros, seqüencialmente. Como o tipo `char` em Java é Unicode (16 bits), é necessário traduzi-lo para algo compatível com ASCII no lado nativo. Neste exemplo utilizamos um *type cast* de `jchar` (16 bits, Unicode) para `char` (8 bits, ASCII) para truncar um caractere de 16 bits para outro de 8 bits. O uso de *type cast* funciona somente se os caracteres forem ASCII puros (valores decimais de 32 a 127) os demais caracteres Unicode gerarão gráficos diferentes quando impostos para ASCII.

O código fonte Java é:

```
/*
 * Implementacao Java do exemplo 1:
 * Passando tipos primitivos como parametros para a implementacao nativa.
 */

class exemplo1
{
  public int    i;
  public char   c;

  public static void main( String[] args )
  {
    exemplo1    app     = new exemplo1();

    app.i = 200000;
    app.c = 'c';

    app.exemplo1Nativo( app.i, app.c );
  }

  private native void exemplo1Nativo( int iParam, char cParam );
```

```
  static
  {
    System.loadLibrary( "exemplo1" );
  }
}
```

O código-fonte C é:

```
/*
 * Implementacao nativa do exemplo 1 em JNI:
 * Passando tipos primitivos como parametros de Java para C. O tipo jchar
 * (16 bits) e' truncado para 8 bits por meio de um type cast.
*/
#include <stdio.h>
#include "exemplo1.h"

JNIEXPORT void JNICALL Java_exemplo1_exemplo1Nativo
  ( JNIEnv* env, jobject jThis, jint iParam, jchar cParam )
{
  char  c;

  c = ( char ) cParam;
  printf( "Implementacao Nativa recebeu como parametros: %d e '%c'\r\n",
          iParam, c );
}
```

O `include` gerado pelo `javah` é:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class exemplo1 */

#ifndef _Included_exemplo1
#define _Included_exemplo1
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     exemplo1
 * Method:    exemplo1Nativo
 * Signature: (IC)V
 */
JNIEXPORT void JNICALL Java_exemplo1_exemplo1Nativo
  (JNIEnv *, jobject, jint, jchar);

#ifdef __cplusplus
}
#endif
#endif
```

Preparando o ambiente Linux, compilando e executando a classe `exemplo1`, obtemos o seguinte resultado:

```
$ LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:.
$ java exemplo1
Implementacao Nativa recebeu como parametros: 200000 e 'c'
$
```

4.2. Exemplo 2: *String* como parâmetro

Com este exemplo ilustramos o uso de *strings* passadas por Java à implementação nativa.

Aqui a classe Java importa uma implementação nativa que imprime uma *string* recebida como parâmetro.

O código fonte Java é:

```
/*
 *   Implementacao Java do exemplo 2:
 * Passando strings como parametros para a implementacao nativa.
 */
class exemplo2
{
  public String s;

  public static void main( String[] args )
  {
    exemplo2    app     = new exemplo2();

    app.s = "Java-JNI";
    app.exemplo2Nativo( app.s );
  }

  private native void exemplo2Nativo( String sParam );

  static
  {
    System.loadLibrary( "exemplo2" );
  }
}
```

O código fonte C é:

```
/*
 * Implementacao nativa do exemplo 2 em JNI:
 * Passando strings como parametros de Java para C.
 */
#include <stdio.h>
#include "exemplo2.h"

JNIEXPORT void JNICALL Java_exemplo2_exemplo2Nativo
  ( JNIEnv* pEnv, jobject jThis, jstring sParam )
{
  const char*   pStr;

  pStr = ( *pEnv )->GetStringUTFChars( pEnv, sParam, 0 );

  if ( pStr != NULL )
  {
    printf( "Implementacao Nativa recebeu a string: %s\r\n", pStr );
  }
  else
  {
    printf( "Implementacao Nativa nao conseguiu adquirir string de Java "
            "(falta memoria?)\r\n" );
  }

  ( *pEnv )->ReleaseStringUTFChars( pEnv, sParam, pStr );
}
```

O `include` gerado pelo `javah` é:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class exemplo2 */

#ifndef _Included_exemplo2
#define _Included_exemplo2
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     exemplo2
 * Method:    exemplo2Nativo
 * Signature: (Ljava/lang/String;)V
 */
JNIEXPORT void JNICALL Java_exemplo2_exemplo2Nativo
  (JNIEnv *, jobject, jstring);
```

```
#ifdef __cplusplus
}
#endif
#endif
```

Executando a classe *exemplo2*, obtemos o seguinte resultado:

```
$ java exemplo2
Implementacao Nativa recebeu a string: Java-JNI
```

4.3. Exemplo 3: Vetor de tipo primitivo como parâmetro

Neste exemplo ilustramos o tratamento de vetores primitivos como parâmetros, tanto com cópia completa como com cópia de apenas um pedaço.

A classe Java que importa uma implementação nativa que, ao receber o vetor primitivo adquire uma cópia completa, listando o seu conteúdo na tela e, em seguida, uma busca cópia parcial, também listando o conteúdo na tela.

O código fonte Java é:

```
/*
 * Implementacao Java do exemplo 3:
 * Passando vetores primitivos como parametros para a implementacao nativa.
 */
class exemplo3
{
  public        int vtInt[]     = new int[ 6 ];
  private       int ix;

  public static void main( String[] args )
  {
    exemplo3    app     = new exemplo3();

    /* Preenche o vetor decrescentemente */
    for( app.ix = 0; app.ix < app.vtInt.length; app.ix++ )
    {
      app.vtInt[ app.ix ] = app.vtInt.length - app.ix;
    }

    app.exemplo3Nativo( app.vtInt );
  }

  private native void exemplo3Nativo( int[] vtIntParam );

  static
  {
    System.loadLibrary( "exemplo3" );
  }
}
```

O código fonte C é:

```
/*
 * Implementacao nativa do exemplo 3 em JNI:
 * Passando vetores primitivos como parametros de Java para C.
 */
#include <stdio.h>
#include "exemplo3.h"

JNIEXPORT void JNICALL Java_exemplo3_exemplo3Nativo
  ( JNIEnv* pEnv, jobject jThis, jintArray vtIntParam )
{
  int   ix;
  int   iArrayLen;
  int   iNumElems;
  int   iDesloc;
  int*  pInt;
  jint* vtLocal;
```

```
  iArrayLen = ( *pEnv )->GetArrayLength( pEnv, vtIntParam );

  vtLocal = ( *pEnv )->GetIntArrayElements( pEnv, vtIntParam, 0 );

  if ( vtLocal != NULL )
  {
    printf( "Implementacao Nativa recebeu um vetor com %d elementos:\r\n",
            iArrayLen );

    for( ix = 0; ix < iArrayLen; ix++ )
    {
      printf( "%d: %d\r\n", ix, vtLocal[ ix ] );
    }
  } else
  {
    printf( "Implementacao Nativa nao pode receber o vetor.\r\n" );
  }

  ( *pEnv )->ReleaseIntArrayElements( pEnv, vtIntParam,
       vtLocal, 0 );

  /* Determina que a parte a ser capturada tem metade do vetor */

  iNumElems = iArrayLen / 2;

  /* E que e' a segunda metade */

  iDesloc = iArrayLen - iNumElems;

  pInt = ( int* ) malloc( sizeof( int ) * iNumElems );

  if ( pInt == NULL )
  {
    printf( "Faltou memoria para alocar espaco para parte do vetor.\r\n" );
    exit( 1 );
  }

  ( *pEnv )->GetIntArrayRegion( pEnv, vtIntParam, iDesloc, iNumElems, pInt );

  printf( "Area capturada do vetor comecando em %d e de tamanho %d:\r\n",
        iDesloc, iNumElems );

  for( ix = 0; ix < iNumElems; ix++ )
  {
    printf( "%d: %d\r\n", ix, pInt[ ix ] );
  }

  free( pInt );
}
```

O `include` gerado pelo `javah` é:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class exemplo3 */

#ifndef _Included_exemplo3
#define _Included_exemplo3
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     exemplo3
 * Method:    exemplo3Nativo
 * Signature: ([I)V
 */
JNIEXPORT void JNICALL Java_exemplo3_exemplo3Nativo
  (JNIEnv *, jobject, jintArray);

#ifdef __cplusplus
}
```

```
#endif
#endif
```

Executando a classe `exemplo3`, obtemos o seguinte resultado:

```
$ java exemplo3
Implementacao Nativa recebeu um vetor com 6 elementos:
0: 6
1: 5
2: 4
3: 3
4: 2
5: 1
Area capturada do vetor comecando em 3 e de tamanho 3:
0: 3
1: 2
2: 1
$
```

4.4. Exemplo 4: Vetor de objetos como parâmetro

Este exemplo ilustra tratamento de vetores de objetos passados como parâmetros à implementação nativa.

A Java importa uma implementação nativa que recebe o vetor e lista seu conteúdo na tela. Foi criada também uma classe cuja única finalidade é servir de molde para o vetor de objetos e conter um elemento primitivo a ser examinado pela implementação nativa

O código fonte Java é:

```
/*
 *   Implementacao Java do exemplo 4:
 * Passando vetores de objetos como parametros para a implementacao nativa.
 */
class exemplo4Vetor
{
  public int i;
}

class exemplo4
{
  public static void main( String[] args )
  {
    exemplo4            app     = new exemplo4();
    exemplo4Vetor       vtObj[] = new exemplo4Vetor[ 6 ];
    int                 ix;

    /* Preenche o vetor decrescentemente */
    for( ix = 0; ix < vtObj.length; ix++ )
    {
      vtObj[ ix ]       = new exemplo4Vetor();
      vtObj[ ix ].i     = vtObj.length - ix;
    }

    app.exemplo4Nativo( vtObj );
  }

  private native void exemplo4Nativo
        ( exemplo4Vetor[] vtObjParam );

  static
  {
    System.loadLibrary( "exemplo4" );
  }
}
```

O código fonte C é:

```
/*
 * Implementacao nativa do exemplo 4 em JNI:
 * Passando vetores de objetos como parametros
 * de Java para C.
 */
#include <stdio.h>
#include "exemplo4.h"

JNIEXPORT void JNICALL Java_exemplo4_exemplo4Nativo
  ( JNIEnv* pEnv, jobject jThis, jobjectArray vtObjParam )
{
  int           ix;
  int           iArrayLen;
  jobject       obj;
  jclass        cObj;
  jfieldID      fId;

  iArrayLen = ( *pEnv )->GetArrayLength( pEnv, vtObjParam );

  if ( iArrayLen > 0 )
  {
    printf( "Implementacao Nativa recebeu um vetor com %d elementos:\r\n",
            iArrayLen );

    obj =  ( *pEnv )->GetObjectArrayElement( pEnv, vtObjParam, 0 );
    cObj = ( *pEnv )->GetObjectClass( pEnv, obj );

    if ( cObj == NULL )
    {
      printf( "Erro determinando classe de objeto!\r\n" );
      exit( 1 );
    }

    /* Elemento de nome "i" e assinatura "I" (inteiro) */
    fId = ( *pEnv )->GetFieldID( pEnv, cObj, "i", "I" );

    printf( "0: %d\r\n", ( *pEnv )->GetIntField( pEnv, obj, fId ) );

    for( ix = 1; ix < iArrayLen; ix++ )
    {
      obj = ( *pEnv )->GetObjectArrayElement( pEnv, vtObjParam, ix );

      printf( "%d: %d\r\n", ix, ( *pEnv )->GetIntField( pEnv, obj, fId ) );
    }
  }
  else
  {
    printf( "Implementacao Nativa nao pode receber vetor.\r\n" );
  }

}
```

O `include` gerado pelo `javah` é:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class exemplo4 */

#ifndef _Included_exemplo4
#define _Included_exemplo4
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     exemplo4
 * Method:    exemplo4Nativo
 * Signature: ([Lexemplo4Vetor;)V
 */
JNIEXPORT void JNICALL Java_exemplo4_exemplo4Nativo
  (JNIEnv *, jobject, jobjectArray);
```

```
#ifdef __cplusplus
}
#endif
#endif
```

Executando a classe `exemplo4` obtemos o seguinte resultado:

```
$ java exemplo4
Implementacao Nativa recebeu um vetor com 6 elementos:
0: 6
1: 5
2: 4
3: 3
4: 2
5: 1
$
```

4.5. Exemplo 5: Retorno de valores de tipo primitivo

Neste exemplo ilustramos o retorno de tipos primitivos da implementação nativa para a classe Java.

A classe Java chama a função nativa que escolhe um valor inteiro aleatório, mostra este valor na tela e retorna este valor ao sair. A classe Java recebe este valor e também o mostra na tela.

O código fonte Java é:

```
/*
 *   Implementacao Java do exemplo 5:
 * Receber valor de tipo primitivo da implementacao nativa.
 */
class exemplo5
{
  public static void main( String[] args )
  {
    exemplo5            app     = new exemplo5();
    int                 i;

    i = app.exemplo5Nativo( );

    System.out.println( "Implementacao Java recebeu valor: "
        + i );
  }

  private native int exemplo5Nativo( );

  static
  {
    System.loadLibrary( "exemplo5" );
  }
}
```

O código fonte C é:

```
/*
 * Implementacao nativa do exemplo 5 em JNI:
 * Retornar valor de tipo primitivo para Java.
 */
#include <stdio.h>
#include "exemplo5.h"

JNIEXPORT jint JNICALL Java_exemplo5_exemplo5Nativo
  ( JNIEnv* pEnv, jobject jThis )
{
  jint          i;

  i = random( );
```

```
  printf( "Implementacao nativa escolheu o numero: %d\r\n", i );

  return i;
}
```

O `include` gerado pelo `javah` é:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class exemplo5 */

#ifndef _Included_exemplo5
#define _Included_exemplo5
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     exemplo5
 * Method:    exemplo5Nativo
 * Signature: ()I
 */
JNIEXPORT jint JNICALL Java_exemplo5_exemplo5Nativo
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Executando a classe *exemplo5* obtemos o seguinte resultado:

```
$ java exemplo5
Implementacao nativa escolheu o numero: 1804289383
Implementacao Java recebeu valor: 1804289383
```

4.6. Exemplo 6: Retorno de *strings*

Neste exemplo ilustramos como retornar uma *string* nova para Java a partir de uma *string* nativa.

A implementação nativa pedirá ao usuário que digite uma *string*. Após gerará uma *string* Java a partir da *string* digitada e retornará esta *string* para o objeto Java.

O código fonte Java é:

```
/*
 *   Implementacao Java do exemplo 6:
 * Receber strings de implementacoes nativas.
 */
class exemplo6
{
  public static void main( String[] args )
  {
    exemplo6            app     = new exemplo6();
    String              s;

    s = app.exemplo6Nativo( );

    System.out.println( "Implementacao Java recebeu valor: " + s );
  }

  private native String exemplo6Nativo( );

  static
  {
    System.loadLibrary( "exemplo6" );
  }
}
```

O código fonte C é:

```
/*
 * Implementacao nativa do exemplo 6 em JNI:
 * Retornar strings para Java.
 */
#include <stdio.h>
#include <fcntl.h>
#include "exemplo6.h"

JNIEXPORT jstring JNICALL Java_exemplo6_exemplo6Nativo
  ( JNIEnv* pEnv, jobject jThis )
{
  char                  vtStr[ 81 ];
  jstring               jStr;
  int                   iFlags;

  /* Por alguma razao, a JVM as vezes configura a entrada padrao
   * nao-bloqueavel, o que significa nao esperar ate' que o usuario
   * digite algo. Precisamos ter certeza de ter a entrada padrao configurada
   * de modo que espere pelo usuario.
   *    A forma de se fazer isso e' usar a funcao fcntl( ) para remover o
   * sinal (flag) de nao-bloqueavel.
   */
  iFlags = fcntl( 0, F_GETFL, 0 );
  fcntl( 0, F_SETFL, iFlags & ~FNONBLOCK );

  printf( "Implementacao nativa, digite uma string: " );
  fgets( vtStr, 80, stdin );

  jStr = ( *pEnv )->NewStringUTF( pEnv, vtStr );

  return jStr;
}
```

O `include` gerado pelo `javah` é:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
+#include <jni.h>
/* Header for class exemplo6 */

#ifndef _Included_exemplo6
#define _Included_exemplo6
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     exemplo6
 * Method:    exemplo6Nativo
 * Signature: ()Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_exemplo6_exemplo6Nativo
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Executando a classe *exemplo6* obtemos o seguinte resultado:

```
$ java exemplo6
Implementacao nativa, digite uma string:
```

ao que se digitou: “testando passagem de strings”,

```
Implementacao Java recebeu valor: testando passagem de strings
```

4.7. Exemplo 7: Objetos como parâmetros

Ilustramos agora a passagem de objetos como parâmetro. Neste exemplo serão utilizados não só elementos do objeto como também será executado um de seus métodos.

Foram implementadas duas classes Java. Uma que contém um elemento e um método que efetua a soma de dois números retornando o resultado, e outra que importa uma implementação nativa que, ao receber o objeto da primeira classe, primeiro mostrará o conteúdo do elemento deste objeto, e depois utilizará o método do objeto para somar um número a este elemento e escreverá o resultado neste elemento mesmo. Finalmente o objeto de interface mostrará o valor deste elemento quando a função nativa retornar.

Seria possível lidar com elementos e métodos da própria classe que chama a implementação nativa através do primeiro parâmetro *jobject* passado ao código nativo da mesma forma como foi feito para com o objeto extra passado. Bastaria apenas utilizar tal parâmetro. Como este exemplo também visa ilustrar a passagem de outros objetos como parâmetros, aproveitou-se para tornar o outro objeto útil ao colocar o elemento e o método dentro dele para serem utilizados pela implementação nativa.

O código fonte Java é:

```
/*
 *   Implementacao Java do exemplo 7:
 *   Passar objetos como parametros para a implementacao nativa.
*/
class exemplo7Parametro
{
  public int            i;

  public int            soma( int num1, int num2 )
  {
    return num1 + num2;
  }
}

class exemplo7
{
  public static void main( String[] args )
  {
    exemplo7            app     = new exemplo7();
    exemplo7Parametro   obj     = new exemplo7Parametro();

    obj.i = 200000;
    app.exemplo7Nativo( obj );

    System.out.println( "Implementacao Java, valor do elemento: "
                        + obj.i );
  }

  private native void exemplo7Nativo( exemplo7Parametro oParam );

  static
  {
    System.loadLibrary( "exemplo7" );
  }
}
```

O código fonte C é:

```
/*
 * Implementacao nativa do exemplo 7 em JNI:
 * Receber parametros objeto de Java.
 */
#include <stdio.h>
```

```
#include "exemplo7.h"

JNIEXPORT void JNICALL Java_exemplo7_exemplo7Nativo
  ( JNIEnv* pEnv, jobject jThis, jobject oParam )
{
  int                   i;
  int                   iElem;
  jfieldID              fId;
  jmethodID             mId;
  jclass                cObj;

  cObj  = ( *pEnv )->GetObjectClass( pEnv, oParam );

  /* Elemento de nome "i" e assinatura "I" */

  fId   = ( *pEnv )->GetFieldID( pEnv, cObj, "i", "I" );
  iElem = ( *pEnv )->GetIntField( pEnv, oParam, fId );

  printf( "Implementacao nativa, valor do elemento: %d\r\n", iElem );

  /* Metodo de nome "soma" e assinatura "(II)I" */

  mId = ( *pEnv )->GetMethodID( pEnv, cObj, "soma", "(II)I" );

  /* Soma "1" a "iElem" */

  i = ( *pEnv )->CallIntMethod( pEnv, oParam, mId, iElem, 1 );

  printf( "Implementacao nativa, metodo \"soma\" com parametros %d e %d"
          " retornou: %d\r\n", iElem, 1, i );

  ( *pEnv )->SetIntField( pEnv, oParam, fId, i );
}
```

O `include` gerado pelo `javah` é:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class exemplo7 */

#ifndef _Included_exemplo7
#define _Included_exemplo7
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     exemplo7
 * Method:    exemplo7Nativo
 * Signature: (Lexemplo7Parametro;)V
 */
JNIEXPORT void JNICALL Java_exemplo7_exemplo7Nativo
  (JNIEnv *, jobject, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Executando a classe *exemplo7* obtemos o seguinte resultado:

```
$ java exemplo7
Implementacao nativa, valor do elemento: 200000
Implementacao nativa, metodo "soma" com parametros 200000 e
1 retornou: 200001
Implementacao Java, valor do elemento: 200001
$
```

5. Epílogo

Neste artigo apresentamos os conceitos e técnicas básicas para o desenvolvimento de aplicações híbridas envolvendo Java a interface nativa Java (JNI) e bibliotecas dinâmicas nativas criadas a partir de código C. Exemplos em C++ foram desconsiderados, pois diferenciam-se de seus contrapartes em C somente por pequenos ajustes sintáticos. Além de apresentarmos as técnicas básicas para implementar tais aplicações, foram apresentados, ainda diversos exemplos que ilustram as diferentes modalidades de interface. Todos os exemplos foram experimentados em plataformas Linux / gcc / JDK 1.3 e Windows (NT e 2000) / MS-Visual C / C++ / JDK 1.3.

Bibliografia e webliografia

[1] Liang, S.; *The JavaTM Native Interface: Programmer's Guide and Specification*; Addison Wesley Longman; Junho de 1999

Indicado pela Sun como a fonte definitiva de informações para utilizar JNI.

[2] http://java.sun.com/docs/books/tutorial/native1.1/
Tutorial da Sun sobre a utilização de JNI. Contém o principal sobre o assunto incluindo exemplos, como fazer e como não fazer.

[3] http://java.sun.com/docs/books/tutorial/native1.1/summary)
Mapping Between JNI and Native Types

[4] http://java.sun.com/products/jdk/faq/jnifaq.html
Frequently Asked Questions sobre JNI, também da Sun.

[5] http://wwwusers.rdc.puc-rio.br/poyart/jdk1.1.8/docs/guide/jni/spec/jniTOC.doc.html

Especificações de JNI em formato técnico. Contém a teoria em que foram baseadas as especificações e até alguns exemplos. Faz parte de um arquivo compactado disponível no endereço da Sun que contém uma grande parte da documentação de Java. O que ele não contém foi mencionado na forma de *link* para o endereço da Sun.

[6] http://java.sun.com/products/jdk/1.1/
Onde conseguir a documentação mencionada acima.

Índice remissivo