JNIEasy

Manual

V1.2.1

Doc. version 1.0

March 31, 2008

Jose Maria Arranz Santamaria

TABLE OF CONTENTS

1. LEGAL

Author: Jose Maria Arranz Santamaria

REPORT: You may wish to report any ambiguities, inconsistencies or inaccuracies you may find in connection with your of evaluation or legal use of the Document ("Feedback"). To the extent that you provide Innowhere with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Innowhere a perpetual, nonexclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose related to the Document and future versions, implementations, and test suites thereof.

GENERAL TERMS: Any action related to this Agreement will be governed by the Spanish law and international copyright and intellectual property laws and that unauthorized use may subject you to civil and criminal liability. The Document is subject to Spanish export control laws and may be subject to export or import regulations in other countries. Licensee agrees to comply strictly with all such laws and regulations and acknowledges that it has the responsibility to obtain such licenses to export, re-export or import as may be required after delivery to Licensee. Neither party may assign or otherwise transfer any of its rights or obligations under this Agreement, without the prior written consent of the other party, except that Innowhere may assign this Agreement to an affiliated company. This Agreement is the parties' entire agreement relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, conditions, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification to this Agreement will be binding, unless in writing and signed by an authorized representative of each party.

2. INTRODUCTION

2.1 THE PROBLEM

"What platform and/or language can I use for my new application?" This is a typical question before starting a new development project. Of course there are many options, but if you go through mature and quality roads there are not so much options.

Java is a prominent and successful language and platform, and perhaps the dominant platform today to develop enterprise-quality applications. Java promises full portability around platforms following the WORA (Write Once Run Anywhere) mantra of Sun.

Anyway, Java is not alone, Java runs on top of well established operating systems like Windows and unixes (Linux, Mac, Solaris…), these operating systems were conceived, of course, before Java, and tons of libraries, applications and services have been developed before Java starts to shine. Sometimes the WORA is not fully possible if you must deal with the specific issues of the platform, or use well established libraries not found equivalent on Java. If your application goes beyond the typical CRUD (Create, Retrieve, Update and Delete) of a relational database using web, usually when developing desktop applications, is common to found "stopper features", integration problems not already solved with Java. These stopper features invite to reevaluate Java as the most appropriate platform to this type of applications (C#?, C/C++?, Windows only tools?, Gtk+?, Qt?). Another scenario is an old but mature application developed in C/C++, the cost of a full rewrite in Java may be enormous, another option is to extend the application following the successful Java route.

Java offers a path to deal with the platform: the JNI (Java Native Interface). The JNI is a C/C++ API working as a gate to/from Java.

But JNI has many problems:

- Is a C/C++ based API, new skills needed.
- Hard to learn, code, read and maintain.
- Error prone (inherent problems of "native compiled" languages).
- Memory issues (memory leaks, crashes).
- Data conversions.
- Strange double modeling: Java classes are object oriented, the JNI methods are not and must deal with C and C++ structures and classes. Synchronization issues.
- Reduction of Java code in favor of native code: portability issues.
- Java classes with native methods and C++ "glue code" usually are not object oriented.
- Establishment of two development/deployment environments (IDEs, configurations …): Java and C/C++.

2.2 THE JNIEASY/JAVA NATIVE OBJECTS SOLUTION

JNIEasy is a new JNI, Java Native Interface, but with no C/C++ code. This is not new; other tools have done this before, then… what is new?

JNIEasy starts a new software paradigm that we can call "**Java Native Objects**" or JNO. JNIEasy takes the main idea of the Java Data Objects (JDO) world to the world of native programming: **transparency**.

JNIEasy converts a POJO (Plain Old Java Object) into a Native Object:

- an object instance is equivalent to a native structure or C++ instance
- a String object may be seen as a native string
- an `Object` reference is equivalent to a native pointer
- fields of a POJO class are equivalent to fields of a native structure or class
- a method call in a POJO is a call of a C or C++ method
- a Java method may be called from native as a callback using a method pointer without JNI conventions.
- a Java object representing a native instance may free its related native memory automatically when is garbage collected reducing (or avoiding) memory leaks.

All is done (as possible) without special classes/artifacts which mimic the native world, using directly `String`, `StringBuffer`, `int`, `long`, `double`, `Method`, `int[]`, `String[]` user defined classes (without special data types) etc.

JNIEasy is **Java centric**, brings the native world to Java, not Java to the native world (the typical approach), *the main purpose of transparency makes Java native programming no different to normal programming in Java*. Alike JDO/Hibernate/JPA with the persistent programming, *JNIEasy makes the native programming a pleasure*, resolving the integration tasks with the underlying platform without JNI or C/C++ (Java access from C/C++ using JNIEasy is optional).

This "magic" is achieved using a technique typically used in JDO and AOP (Aspect Oriented Programming): *bytecode enhancement* (or bytecode weaving). The bytecode enhancement extends a normal compiled class with new capabilities (new code) but keeping the source code the same avoiding intrusive and disturbing artifacts used in the traditional approach to the problem. The bytecode enhancement enables the IOC (Inversion Of Control) paradigm used by the framework, to be fully transparent to the programmer; avoids the use of Java reflection making transparent native programming performant too and may be done in the command line or on class loading (simplifying the development lifecycle).

Although transparency is the main feature of JNIEasy (the "easy" level), the framework offers a very rich set of classes and interfaces to deal with the native world too. The programmer has ever the option of a fine control of the native memory or relay in the framework using Java basic data types (int, Integer, int[], String, StringBuffer, String[], Method … all of them have a "native layout" in JNIEasy), in fact, this rich model is used behind the scenes by the framework to achieve the transparency. Of course these utility classes and interfaces are Java centric too.

Does JNIEasy break the WORA promise?

Yes and no. The WORA promise is "easily" achieved using Java only and cross-platform Java libraries, but these "cross-platform" libraries are not ever developed using Java only, the popular AWT toolkit and many basic standard classes are very dependent on the platform (many native methods), why are they WORA too? because they are ported to almost any

platform and work ("almost") in the same way in all platforms. If the popular SWT toolkit (very platform dependent) was present in all Java installations, it would be considered part of the WORA.

If your project needs a special feature present in the underlying platform, do you must wait to Sun (and others) to deliver a WORA solution? , this is not reasonable and impossible if your need is an integration issue with another specific tool.

JNIEasy helps you to construct your own "native WORA in Java": your Java code may detect the underlying platform and use a specific Java code to this platform, and offer a high level cross-platform API, your customers/users do not need to select a specific platform version of your program, because no custom DLL is needed to be delivered with your code.

JNIEasy offers several cross-platform options like the variable size primitives, for instance: a Java `long` can be reflected in the native side as a 32 bits, or 64 bits integer depending on the running platform (a `long` can hold a cross-platform address), a Java `char` can be a C `char` (1 byte), a Win32 `wchar_t` (2 bytes) or a gcc's `wchar_t` (4 bytes) in unixes (Linux, Mac OS X, Solaris…), in a extreme a Java field can be 0 native size in a specific platform and not zero in another platform. The objective is simple: one Java layout-multiple native layouts, the multiple native layouts usually are due to the use of C macros, to achieve this type of "flexible memory selection" based on the platform JNIEasy uses a conditional "C inspired" macro system too. This macro system is used to select the appropriate name of dynamic libraries too (e.g. `MSVCRT.dll` on Windows and `libc.so.6` on Linux, `libc.dylib` on Mac OS X, `libc.so.1` on Solaris) or to resolve the problems of different name mangling of C++ compiler, the correct native exported name is selected using this macro system.

Never the WORA was so easy and powerful… and "so Java"…

3. CONSIDERATIONS

3.1 SCOPE

This manual makes an extensive documentation of JNIEasy features but must be complemented with the API documentation in javadoc format.

3.2 REQUIRED USER TECHNICAL SKILLS

JNIEasy avoids coding with JNI and C/C++, but basic or medium knowledge of C or C++ is needed, because a native API exported in a dynamic library uses C/C++ types and conventions.

Only C/C++ languages are considered in the native size because they are the most popular "native" languages and JNI is defined only to C/C++. JNIEasy tries to avoid JNI use at all, furthermore native code can access (optionally) to Java code without JNI, wherefore other native languages like FORTRAN or Delphi can be used with JNIEasy because these languages can create and bind Dynamic Link Libraries/shared objects; the developer must convert "mentally" C/C++ data types to the selected language and vice versa to understand the JNIEasy's native operations and structures.

3.3 TECHNICAL REQUIREMENTS, LIMITATIONS AND DEPENDENCIES

The Java part of JNIEasy is compiled and tested with Sun's Java Standard Edition v1.4. It has been tested with v1.5 (Java SE 5) with no problem.

Four Intel x86/32 bit (IA-32) platforms are supported at this moment: Windows, Linux, Mac OS X (since 10.4) and Solaris with Pentium Pro/i686 as the minimum processor:

- Windows: `JNIEasy.dll` and `JNIEasy.lib` (Windows) are compiled with Visual C++ 6.0, Win32 DLL target and Pentium Pro processor
- Linux: `libJNIEasy.so` is compiled with gcc-4.0.2 using glibc-2.3.2
- Mac OS X: `libJNIEasy.jnilib` is compiled with gcc-4.0, provided compiler of XCode 2.5 on a Mac OS X 10.4.5 Tiger Intel (10.5 version, Leopard, is reported to work too)
- Solaris: `libJNIEasy.so` is compiled with the native gcc-3.4.3 compiler on OpenSolaris 11 build 85 (Solaris 11, SunOS 5.11)

Java code is compiled with Java Virtual Machine v1.4.

User defined native capable classes can be compiled with Java SE SDK v1.5 and upper and use the new language features introduced in Java SE 5.0 (generics, enums, variable args etc), but a native capable class can not be a generic class or contain native fields or native method signatures with the new language features, in brief: native "enabled" types do not support

language features of Java SE 5.0. There is no limitation in normal classes using user defined native capable classes.

JNIEasy can map C/C++ exported methods, structures, unions and classes, but C++ templates may be exported but not bound to Java generics.

JNIEasy depends on Javassist (LGPL, http://www.csg.is.titech.ac.jp/~chiba/javassist/) and Bouncy Castle (http://www.bouncycastle.org/licence.html).

3.4 DOCUMENT CONVENTIONS

A Verdana font is used to describe the JNIEasy architecture.

```
A Courier New font is used to Java, C/C++ and XML code fragments.
```

4. ARCHITECTURE

4.1 DEFINITIONS

4.1.1 Native

Represents the C/C++ world, Java elements declared as native means they are related with the C/C++ side.

4.1.2 Native memory

Is memory addressed and controlled by the native side (or by JNIEasy in the Java side), a block of native memory is created with `malloc()` (or similar methods), or with `new` (generally this C++ sentence is based on `malloc()`). Native memory must be distinguished from "Java memory", Java memory is the memory controlled exclusively by the Java Virtual Machine (JVM). A JNIEasy main task is synchronizing native memory and Java memory transparently.

4.1.3 Native capable

Any Java element that may have a role in JNIEasy, prepared to represent something in the native side. Native capable may be classes, methods, fields, object instances…

4.1.4 Native capable class

A native capable class is a normal Java class previously that implements the `NativeCapable` interface, prepared to represent *native object instances*. There are two types: predefined classes (provided by the framework) and user defined (enhanced).

4.1.5 Native object instance

A native object instance is a Java object of a native capable class with a related native memory block, the native memory layout can be a C structure, a C++ class, a C native string ('\0' terminated), a simple native integer etc. A native instance always has a native memory address (different to Java memory address, this address is managed by the JVM not JNIEasy). A native object instance is reflected in the native side as a class/structure/union instance, a pointer to method, a native array (int[]), a zero ended string etc.

4.1.6 Native capable object instance

An object instance of a native capable class that is not native (no native address/native memory block associated). A native capable object instance can be made native.

4.1.7 Native interfaces

Are interfaces predefined by JNIEasy inheriting from `NativeCapable`. Represent different types of native elements: `Structure` represents a C structure, `CPPClass` a C++ class, `NativeString`

a native string mapped with `java.lang.String`, `NativeIntegerArray` a native array mapped with a normal `int[]` Java array etc. The developer does not need deal with these interfaces unless a fine control is requested.

4.1.8 User defined native capable class

Is a class coded by the user and enhanced or prepared to be enhanced on load time. The enhancement process makes this class native capable implementing the `NativeCapable` interface. The fields and methods of these classes can be declared as native, and managed by the framework. The code of a user defined native capable class is *managed code* by the enhancer. The enhancement provides the native capabilities to the user class.

4.1.9 "Can be native" class/object

A "can be native" class is a basic standard Java class with a native corresponding element. For instance: `String` and `StringBuffer` objects correspond to a `'\0'` terminated native string (`char*` if ANSI or `wchar_t*` if UNICODE), an `int[]` array corresponds to a native `jint[]` array (`jint` usually is equivalent to `int` C data type), a `String[]` or `StringBuffer[]` with an array of `char*` or `wchar_t*` pointers, an `Integer` object with an addressed `jint` integer, a `java.lang.reflect.Method` reference with a native method pointer etc.

4.1.10 Predefined native capable class

Is a native capable class predefined by JNIEasy. JNIEasy provides a rich set of native capable classes representing (wrapping) most of the basic native elements like strings, arrays of primitive types etc. Most of these classes are artifacts to make native *can be native objects*, an instance of a predefined class typically wraps a "can be native" object. These classes may be not used directly, because the wrapped *can be native objects* may be used instead (in managed/enhanced code). Furthermore, predefined native classes are hidden by the framework and cannot be used explicitly, instead, every predefined class has a public native interface (`NativeString`, `NativeIntegerObject` etc), object instances of theses classes are created using the framework and may be accessed using the related native interface. Example: a `NativeString` object wraps a `String` object.

Predefined native capable classes/interfaces provides you with a fine control of the native memory, they are used extensively and internally to provide the native behavior of user defined native capable classes with "can be native" fields (`String`, `Method`, `Integer` etc), and methods declared as native proxies, notwithstanding, they can be used in native fields too (a `NativeString` field is basically the same as a `String` field, but with `NativeString` is possible to access a specific character without fetch/unfetch the full native string from/to native).

Use of predefined native capable classes makes the distinction of a fully transparent native programming (using POJOs with extensive use of code generation and/or enhancement) where the Java code is like C/C++ programming in Java with all Java advantages (the "easy" level), or using predefined native capable classes to gain more control of the native memory and code no so transparent and clean.

4.1.11 Native enabled Java type

A "native enabled Java type" is a Java type that can represent a native type. Native enabled types are: native capable classes and interfaces (user defined and predefined), "can be native" classes and primitive types.

4.1.12 Native (capable) wrapper class/object

A native capable wrapper class is a predefined or user defined native capable class whose main purpose is to make native a contained (wrapped) "can be native" object. The user can use these classes directly to take full control of the native memory associated to the "can be native" object or use directly the "can be native" object in a managed code (a native wrapper object will be used behind the scenes). Example: the predefined class implementing `NativeString` wraps `String`, `NativeIntegerObject` wraps `Integer` and so on.

4.1.13 Native type

Is the "native view" of a native enabled Java type usually contained in a native instance (e.g. an `int` field declared as native has a native type). Defines how is made native an instance of this type. For example, a native object of a class implementing the `NativeString` or a `String` object may represent a '\0' terminated string of `char` or a L'\0' terminated string of `wchar_t`, depending on the native type associated to this instance (used to create the instance or defaulted if used `new`), the `NativeString` interface and `String` class are not sufficient themselves to describe the native layout. The native type may be seen as the type of the native object itself (the type of the resulting of `new MyNativeClass()`), not the type of a reference to this object (ex. `MyNativeClass ref`). Any native capable object (non-native) has a concrete native type, this native type will be used to define the native layout when native. A native capable or "can be native" class usually has a default, complete, native type (there are exceptions like `Method` and related predefined native capable classes, the default native type is incomplete, a native method signature must be specified).

4.1.14 Native variable type

Represents a native declaration of a field member with a native enabled type, a formal parameter declaration or return type of a Java method declared as native. It must be seen as the native type of the declaration of a variable. In this Java field declaration: `MyNativeClass ref = new MyNativeClass()`, the native view of the created object instance is driven by the *native type*, and the native view of the reference is driven by the *native variable type*. Native variable types mimic the C conventionalism of "by pointer" or "by value", if "by pointer" `ref` is a pointer to the created object on native memory and may change, if "by value" `ref` "contains" the native memory itself and cannot change (the implicit address); this distinction is specially useful declaring method parameters and fields "by value" or "by pointer", for instance: a int[] Java field of a user defined native capable class (a C++ class or structure or union in the native side) can be declared "by value" (the array is embedded) or "by pointer" (the field is a pointer to array, int*). The "by reference" (using the `&` sign of C) is a special case of "by pointer" and is not defined by JNIEasy.

4.1.15 Native capable method signature

When a Java method has parameters and return using native enabled types.

4.1.16 Native method signature

Represents the JNIEasy concrete native declaration of parameters and return of a native capable method signature (of a DLL method reflected in Java, a Java callback…). The native call convention, standard call or C call, is declared too.

4.1.17 Native (capable) field

A native capable field is a normal Java field member with a native enabled type. If declared as native this field matches with a hypothetical field in the native layout of a native instance of the container class, the native layout may be a primitive data type, a pointer (if declared "by pointer"), a native structure, class or array (if declared "by value"). A native field is a native capable field of a native instance and has an implicit address.

4.1.18 Native (capable) method

A native capable method is a normal Java method with a native capable signature, declared as native using a native method signature. This method may be declared as a proxy of a method contained in a Dynamic Link Library (DLL[1]) or a callback "callable" from C/C++ and may represent a C++ constructor, a C++ method (instance method), a C method or a field wrapped with a special method. A native capable method is not necessarily contained in a native capable class. A Java native method is a native capable method with a native address and can be called from native. A Java native method has a related native object method ever (sometimes is created internally by the framework), this object can be used to manage the method like a pointer to method in Java and pass a reference of this object to the native side (the native side receives this reference like a pointer to method).

4.1.19 Native (capable) field-method

A native capable field-method is a native capable method wrapping a native capable field with the following signature:

Static field

```
FieldType fieldMethodName(int opcode,FieldType value)
```

Instance field

```
FieldType fieldMethodName(ContainerClass obj,
                          int opcode,FieldType value)
```

With the mission of get/set the value of the field; if `opcode` is:

`NativeFieldMethod.GET`, or 0: the method returns the current value of the field, the `value` parameter is ignored.

`NativeFieldMethod.SET`, or 1: the field is updated with the `value` parameter, and returns the new field value.

`NativeFieldMethod.GET_SET`, or 2: the field is updated with the `value` parameter, and returns the previous field value to the updating.

The framework can offer this way to access a native Java field or a C/C++ field directly.

4.1.20 Native (capable) object method

[1] To simplify the DLL, Dynamic Link Library, acronym will be used to identify a Linux based shared object too.

A native (capable) object method is a native (capable) utility object working as a proxy of a Java native capable method or a field seen as a method (native callback) or a DLL exported method or field (native proxy).

4.1.21 Native callback

A native callback is a native object method working as a proxy of a Java native capable method or a field seen as a method. This object makes accessible the Java native capable method (or field) from native, redirects a native call to the wrapped Java method using reflection or a direct call.

4.1.22 Native proxy of a DLL exported method (or field)

A native proxy of a DLL method is a native object method that makes the DLL method callable from Java, converting the Java call to a native call. Usually a Java native capable method can be used as an external and transparent front end using in its implementation the native proxy to call the DLL. In the field case a special method is used to read/write the DLL exported field, the field is seen as a method.

4.1.23 Framework managed code

A code zone is "managed" if it was bytecode enhanced by the framework. Managed code zones are the body of instance methods of user defined native capable classes. In these methods accessing (read/modification) to native declared fields is intercepted by the framework ensuring the native memory is read to get the current "real" value of the field or modified to synchronize the native memory with a new field value. In managed code the native memory manipulation is made indirectly and transparent to the user. Managed methods are not necessarily native (capable) methods.

4.1.24 Enhancement time

User defined classes must be enhanced to be converted in native capable classes. Enhancement can be done on filesystem or on runtime (on class loading). If the enhancement is done on filesystem (usually using the `NativeEnhancerCmd` class in the command line or Ant task), this task does not use the native memory and/or load dynamic link libraries. Running this task may be called "enhancement time".

4.1.25 Generation code time

JNIEasy has a special tool (usually using the `NativeCodeGeneratorCmd` class in the command line or Ant task) to generate the code of Java classes working as a proxy of DLL exported methods, usually C methods. The generated code is used to call from Java DLL methods bound on demand (the first time was used); these classes are especially useful to map DLLs with tons of exported methods. This task does not use native memory but may use the enhancer (on-load mode). Running this task may be called "generation code time".

4.1.26 Runtime

When performing any task involving creation or use of native memory (using native instances) and/or loading DLLs. The most important interface to these tasks is `NativeManager`.

4.1.27 Native transaction

A native transaction is conceptuality very similar to a database transaction, or a Java EE or JDO transaction but with the native memory seen as a "repository". During a native transaction the state of the native memory is saved before any modification and is restored to the original values if the transaction is aborted (rollback).

4.2 FRAMEWORK MODULES

The framework is fully included in the Java package: `com.innowhere.jnieasy.core`

This package is divided in several sub packages; these packages are coincident with the main tasks/parts of JNIEasy: enhancer, code generator, native type declaration, native interfaces, factories, native memory management, transaction management and listeners.

The framework starts with the abstract class `JNIEasy`. The static `get()` method returns a singleton object of this class; using this singleton we can obtain most of the utility classes/interfaces.

```
JNIEasy jnieasy = JNIEasy.get();
```

JNIEasy's public API is based mainly on interfaces; using interfaces helps to keep the API clean (no undocumented and disturbing methods and fields) and robust to internal changes and evolutions, but the reader must keep in mind that most of lower interfaces have a unique implementation class into the framework, for instance, only one class implements `NativeInteger` interface, using the `NativeInteger` data type is not different to use the related internal class.

4.2.1 Enhancer

The enhancer utilities are localized in the package: `com.innowhere.jnieasy.core.enh`

Provides utility classes/interfaces related to bytecode enhancement tasks of native capable classes.

The main interface is `NativeEnhancer`, this interface is implemented by a singleton object that can be obtained with the `JNIEasy.getEnhancer()` method. For instance:

```
NativeEnhancer enh = JNIEasy.get().getEnhancer();
```

With the enhancer object we can enhance the classes (.class files) described with special XML files, and enable/disable the "on load enhancement" feature.

The enhancement task may be launch directly on command line using the executable class `NativeEnhancerCmd`.

4.2.2 Code generator

The code generation utilities are located in the package:

`com.innowhere.jnieasy.core.cgen`

Provides utility classes/interfaces used to generate the code of Java classes working as proxies of exported methods (usually C methods) of big DLLs.

The main interface is `NativeCodeGenerator`, this interface is implemented by a singleton object that can be obtained with the `JNIEasy.getCodeGenerator()` method. For instance:

```
NativeCodeGenerator codeGen = JNIEasy.get().getCodeGenerator();
```

With the code generator object we can generate the Java code of the user defined classes specified in special XML files.

The code generation task may be launch directly on command line using the executable class `NativeCodeGeneratorCmd`.

4.2.3 Native type declaration

The package: `com.innowhere.jnieasy.core.typedec` contains:

1. The native type declaration utility: the interface `NativeTypeManager` implemented by a singleton object that can be obtained with the `JNIEasy.getTypeManager()` method. For instance:

```
NativeTypeManager typeMgr = JNIEasy.get().getTypeManager();
```

2. Native type interfaces representing native type declarations: all of them inherit from `TypeNative`.

3. The variable native type declaration utility: the interface `NativeVarTypeManager` implemented by a singleton object that can be obtained with the `JNIEasy.getVarTypeManager()` method. For instance:

```
NativeVarTypeManager varTypeMgr = JNIEasy.get().getVarTypeManager();
```

4. The variable native type interface `VarTypeNative` representing variable native type declarations.

5. The native method signature declaration utility: the interface `NativeSignatureManager` implemented by a singleton object that can be obtained with the `JNIEasy.getSignatureManager()` method. For instance:

```
NativeSignatureManager sigMgr = JNIEasy.get().getSignatureManager();
```

6. Interfaces of native method signatures: all of them inherit from `NativeBehaviorSignature`.

4.2.4 Native interfaces

The native interfaces are divided in two packages:

1. Package `com.innowhere.jnieasy.core.data`

   Contains all native interfaces implemented by predefined native capable classes starting on `NativeCapable`, with the exception of interfaces related to native methods.

2. Package `com.innowhere.jnieasy.core.method`

   Contains native interfaces related to native methods.

4.2.4.1 Table of data native interfaces with predefined classes

Name	Java class(es) matched[2]	Contained element(s)	User defined classes	Matched native type of a reference	Object size[3]	Notes
CPPClass	Itself and user defined classes as C++ classes	Multiple fields and methods	yes	*CPPClass** if "by pointer", *CPPClass* if "by value"	Depends on declaration	The predefined class has not native fields/ methods (unknown)
NativeArrayOfArray	A multiple dimensional array and user defined classes as multidimensional arrays	A multiple dimensional array	yes	Depends on declaration	Depends on declaration	The predefined class contains an `Object[]` field
NativeBoolean	`boolean`	`boolean` field	no	`jboolean*`	1	
NativeBooleanArray	`boolean[]`	`boolean[]` field	no	`jboolean*, jboolean x[n]`	1* length	
NativeBooleanObject	`Boolean`	`Boolean` field	no	`jboolean*`	1	
NativeBooleanObjectArray	`Boolean[]`	`Boolean[]` field	no	`jboolean**, jboolean* x[n]`	platf.size * length	
NativeByte	`byte`	`byte` field	no	`jbyte*`	1	
NativeByteArray	`byte[]`	`byte[]` field	no	`jbyte*, jbyte x[n]`	1* length	
NativeByteObject	`Byte`	`Byte` field	no	`jbyte*`	1	

[2] This class may be used to create a native capable object implementing the interface. Examples:

```
NativeTypeManager typeMgr = JNIEasy.get().getTypeManager();
TypeCanBeNativeCapable typeStr = (TypeCanBeNativeCapable)typeMgr.dec(String.class);
NativeString strObj = (NativeString)typeStr.wrapValue("any string");
TypeNativeObject typeInt = (TypeNativeObject)typeMgr.dec(NativeInteger.class);
NativeInteger objInt = (NativeInteger)typeInt.newValue();
```

[3] Sizes are calculated with the default values, they can be changed, example: the Java `long` data type can be reflected as a 4 bytes integer (a C `int`), sizes of related data types like `NativeLong`, `NativeLongObject`, `NativeLongArray` etc can be affected.

NativeByteObjectArray	`Byte[]`	`Byte[]` field	no	`jbyte**`, `jbyte* x[n]`	platf.size * length	
NativeCapable	Itself	None	yes (but using inherited interf.)	*`NativeType*`* if "by pointer" ("by value"not allowed)	Unknown	
NativeCapableArray	`NativeCapable[]` classes and inherited and user defined classes as native capable arrays	An inherited `Object[]` field	yes	*`ElementType**`* or *`ElementType* x[n]`* if "by pointer", *`ElementType*`* or *`ElementType x[n]`* if "by value"	platf.size * length if "by pointer" or elem.size* length if "by value"	The predefined class contains an `Object[]` field
NativeCharacter	`char`	`char` field	no	`jchar*`	2	
NativeCharacterArray	`char[]`	`char[]` field	no	`jchar*`, `jchar x[n]`	2* length	
NativeCharacterObject	`Character`	`Character` field	no	`jchar*`	2	
NativeCharacterObjectArray	`Character[]`	`Character[]` field	no	`jchar**`, `jchar* x[n]`	platf.size * length	
NativeDouble	`double`	`double` field	no	`jdouble*`	8	
NativeDoubleArray	`double[]`	`double[]` field	no	`jdouble*`, `jdouble x[n]`	8* length	
NativeDoubleObject	`Double`	`Double` field	no	`jdouble*`	8	
NativeDoubleObjectArray	`Double[]`	`Double[]` field	no	`jdouble**`, `jdouble* x[n]`	platf.size * length	
NativeFloat	`float`	`float` field	no	`jfloat*`	4	
NativeFloatArray	`float[]`	`float[]` field	no	`jfloat*`, `jfloat x[n]`	4* length	
NativeFloatObject	`Float`	`Float` field	no	`jfloat*`	4	
NativeFloatObjectArray	`Float[]`	`Float[]` field	no	`jfloat**`, `jfloat*`	platf.size	

				`x[n]`	* length	
NativeInteger	`int`	`int` field	no	`jint*`	4	
NativeIntegerArray	`int[]`	`int[]` field	no	`jint*`, `jint x[n]`	4* length	
NativeIntegerObject	`Integer`	`Integer` field	no	`jint*`	4	
NativeIntegerObjectArray	`Integer[]`	`Integer[]` field	no	`jint**`, `jint* x[n]`	platf.size * length	
NativeLong	`long`	`long` field	no	`jlong*`	8	
NativeLongArray	`long[]`	`long[]` field	no	`jlong*`, `jlong x[n]`	8* length	
NativeLongObject	`Long`	`Long` field	no	`jlong*`	8	
NativeLongObjectArray	`Long[]`	`Long[]` field	no	`jlong**`, `jlong* x[n]`	platf.size * length	
NativeObjectArray	`Object[]` and inherited	An inherited `Object[]` field	yes (but using inherited interf.)	*`ElementType **`* or *`ElementType * x[n]`* if "by pointer", *`ElementType *`* or *`ElementType x[n]`* if "by value"	platf.size * length if "by pointer" or elem.size* length if "by value"	The predefined class contains an `Object[]` field. User defined classes contains an object inherited array field
NativePointer	Itself and user defined classes as Pointer	An inherited `Object` field	yes	*`ElementType **`*	platf.size	The predefined class contains an Object field
NativeString	`String`	`String` field	no	`char*` or `wchar_t*`	char size* length	Depends on encoding
NativeStringBuffer	`StringBuffer`	`StringBuffer` field	no	`char*` or `wchar_t*`	char size* length	Depends on encoding
NativeShort	`short`	`short` field	no	`jshort*`	2	
NativeShortArray	`short[]`	`short[]` field	no	`jshort*`, `jshort x[n]`	2* length	

NativeShortObject	`Short`	`Short` field	no	`jshort*`	2	
NativeShortObjectArray	`Short[]`	`Short[]` field	no	`jshort**`, `jshort* x[n]`	platf.size * length	
NativeStringAnsi	Itself	`String` field	no	`char*`	1*length	
NativeStringAnsiArray	Itself	`String[]` field	no	`char**`, `char* x[n]`	platf.size * length	
NativeStringArray	`String[]`	`String[]` field	no	`char**` and `char* x[n]` or `wchar_t**`, `wchar_t* x[n]`	platf.size * length	Depends on encoding
NativeStringBufferAnsi	Itself	`StringBuffer` field	no	`char*`	1*length	
NativeStringBufferAnsiArray	Itself	`StringBuffer[]` field	no	`char**`, `char* x[n]`	platf.size * length	
NativeStringBufferArray	`StringBuffer[]`	`StringBuffer[]` field	no	`char**` and `char* x[n]` or `wchar_t**`, `wchar_t* x[n]`	platf.size * length	Depends on encoding
NativeStringBufferUnicode	Itself	`StringBuffer` field	no	`wchar_t*`	2*length	
NativeStringBufferUnicodeArray	Itself	`StringBuffer[]` field	no	`wchar_t**`, `wchar_t* x[n]`	platf.size * length	
NativeStringUnicode	Itself	`String` field	no	`wchar_t*`	2*length	
NativeStringUnicodeArray	Itself	`String[]` field	no	`wchar_t**`, `wchar_t* x[n]`	platf.size * length	
Structure	Itself and user defined classes as structures	Multiple fields and methods	yes	*`Structure*`* if "by pointer", *`Structure`* if "by value"	Depends on declaration	The predefined class has not native fields/ methods (unknown)
Union	Itself and user defined	Multiple fields and methods	yes	*`Union*`* if "by pointer", *`Union`* if "by	Depends on declarati	The predefined class has not

	classes as unions			value”	on	native fields/ methods (unknown)

4.2.4.2 Table of method native interfaces with predefined classes

Name	Java class/method matched	Contained element(s)	User defined classes	Matched native type of a reference	Notes
CFieldMethod	none	none	no	A pointer to a field-method (no instance param.)	Used to map a static field exported in a DLL
CMethod	none	none	no	A pointer to a method (no instance param.)	Used to map a static method exported in a DLL
CPPConstructor	none	none	no	A pointer to a C method working as a constructor	Used to map a C method working as a constructor exported in a DLL
CPPMethod	none	none	no	A pointer to a method (first param. is the instance)	Used to map a instance method exported in a DLL
NativeConstructor	none	none	yes (but using inherited interf.)	A pointer to a C method working as a constructor	
NativeConstructorReflection	`Constructor`	`Constructor` field	no	A pointer to a C method working as a constructor	
NativeConstructorReflectionArray	`Constructor[]`	`Constructor[]` field	no	*`ConstrucPtr*`* or *`ConstrucPtr`* `x[n]`	
NativeDirectConstructorCallback	The mapped constructor	none	yes	A pointer to a C method working as a constructor	The Java class is generated on the fly if necessary
NativeDirectInstanceFieldCallback	The mapped	none	yes	A pointer to a field-method	The Java class is generated on

	instance field			(first param. is the instance)	the fly if necessary
NativeDirectInstanceMethodCallback	The mapped instance method	none	yes	A pointer to a method (first param. is the instance)	The Java class is generated on the fly if necessary
NativeDirectStaticFieldCallback	The mapped static field	none	yes	A pointer to a field-method (no instance param.)	The Java class is generated on the fly if necessary
NativeDirecStaticMethodCallback	The mapped static method	none	yes	A pointer to a method (no instance param.)	The Java class is generated on the fly if necessary
NativeFieldMethodReflection	`Field`	`Field` field	no	A pointer to a field-method	The field may be static or instance. If static the object can be cast to `NativeStaticFieldMethodReflection`, else to `NativeInstanceFieldMethodReflection`
NativeFieldMethodReflectionArray	`Field[]`	`Field[]` field	no	*`MethodPtr`*`*` or *`MethodPtr`* `x[n]`	
NativeInstanceFieldMethod	none	none	yes (but using inherited interf.)	A pointer to a field-method (first param. is the instance)	
NativeInstanceFieldMethodReflection	`Field`	`Field` field	no	A pointer to a field-method (first param. is the instance)	The reflection field object must be non-static.
NativeInstanceMethod	none	none	yes (but using inherited interf.)	A pointer to a method (first param. is the instance)	

NativeInstanceMethodReflection	`Method`	`Method` field	no	A pointer to a method (first param. is the instance)	The reflection method object must be non-static
NativeMethodReflection	`Method`	`Method` field	no	A pointer to a method	The method may be static or instance. If static the object can be cast to `NativeStaticMethodReflection`, else to `NativeInstanceMethodReflection`
NativeMethodReflectionArray	`Method []`	`Method []` field	no	*`MethodPtr*`* or *`MethodPtr`* `x[n]`	
NativeStaticFieldMethod	none	none	yes (but using inherited interf.)	A pointer to a field-method (no instance param.)	
NativeStaticFieldMethodReflection	`Field`	`Field` field	no	A pointer to a field-method (no instance param.)	The reflection field object must be static.
NativeStaticMethod	none	none	yes (but using inherited interf.)	A pointer to a method (no instance param.)	
NativeStaticMethodReflection	`Method`	`Method` field	no	A pointer to a method (no instance param.)	The reflection method object must be static

4.2.5 Utilities

The package: `com.innowhere.jnieasy.core.util` contains a utility interface, `NativeCapableFactory`, used as shortcut to create native capable objects of predefined native capable classes, wrapping other native capable or "can be native" objects. Implemented by a singleton object that can be obtained with the `JNIEasy.getNativeCapableFactory()` method. For instance:

```
NativeCapableFactory objFactory = JNIEasy.get().getNativeCapableFactory();
```

Furthermore it contains two utility classes: `NativeCapableUtil` with tasks related to native capable objects and `NativePrimitiveUtil` to primitive values.

4.2.6 Native memory management

The native memory management tools are located in the package:

`com.innowhere.jnieasy.core.mem`

The main tool is the interface `NativeManager`, used to manage the native lifecycle of native capable objects and other native memory tasks. Is implemented by a singleton object that can be obtained with the `JNIEasy.getNativeManager()` method. For instance:

```
NativeManager natMgr = JNIEasy.get().getNativeManager();
```

Another important tool is the `DLLManager` interface, used to manage DLLs (load/find). Each DLL/shared object is represented with an object implementing the interface `DynamicLibrary`; using this interface, DLL exported methods and fields can be mapped with native proxy methods. The dynamic library of the framework (`JNIEasy.dll` on Windows, `libJNIEasy.so` on Linux and Solaris, `libJNIEasy.jnilib` on Mac OS X) is a special case, it is mapped with the `JNIEasyLibrary` interface. The `DLLManager` is obtained with the `JNIEasy.getDLLManager()` method:

```
DLLManager dllMgr = JNIEasy.get().getDLLManager();
```

And the `JNIEasyLibrary` singleton is obtained with:

```
JNIEasyLibrary fwDll = JNIEasy.get().getJNIEasyLib();
```

4.2.7 Transaction management

The native transaction tools are located in the package:

`com.innowhere.jnieasy.core.txn`

The main interface is `NativeTransaction`, used to begin and end (commit or rollback) a native transaction. The `NativeTransaction` interface is implemented by thread dependent instances, only one transaction may be alive per thread, and the current thread's transaction is returned with the method `NativeManager.currentTransaction()`.

4.2.8 Listeners

The package `com.innowhere.jnieasy.core.listener` provides interfaces and classes used to register callbacks and event listeners of the life cycle of native instances.

There are two types of listeners:

1. Interfaces implemented by native capable classes: the native instance changing its native state receives the matched event. These are the *`EventType`*`Callback` interfaces.

2. Interfaces implemented by normal Java classes working as listeners: instances of these classes are registered as listeners. These are the `InstanceLifecycleListener` based interfaces.

5. DECLARATION AND CREATION OF NATIVE CAPABLE OBJECTS

There are two types of native capable objects: predefined and user defined.

5.1 CREATION OF PREDEFINED NATIVE CAPABLE OBJECTS

The predefined native classes are hidden by the framework and cannot be created explicitly with `new`, related interfaces must be used against, and we need to specify what native interface is used to create a predefined native capable object.

5.1.1 Creation with native types

A native interface of a predefined native capable class is not ever a complete native type. An example is `NativeString`: a native string may be ANSI (`char*`) or UNICODE (`wchar_t*`), a native type may be necessary to create native capable objects of predefined classes.

To declare native types, the `NativeTypeManager` object is the direct way:

```
NativeTypeManager typeMgr = JNIEasy.get().getTypeManager();
```

5.1.1.1 Example: a native capable integer object.

The `NativeInteger` is a final interface (native capable objects can be created) represents native integers (`jint` addressed values); a native object of this type is a `jint` in the native memory, and a Java reference can be seen as a `jint*` pointer. The `TypeNativeObject.newValue()` is used to create an object of the declared class.

```
TypeNativeObject typeInt =

                (TypeNativeObject)typeMgr.dec(NativeInteger.class);

NativeInteger natInt = (NativeInteger)typeInt.newValue();

natInt.setIntValue(10);
```

When this native capable object is made native the 10 value is written on the related native memory (a `jint`).

5.1.1.2 Example: a Unicode native capable string.

```
TypeNativeString typeStr = typeMgr.decString(StringEncoding.UNICODE);
```

Now a native capable string object can be created using UNICODE encoding and wide character size, with the method `TypeCanBeNativeCapable.wrapValue(Object)` or the method `TypeNativeString.newString(String)`:

```
NativeString natStr = (NativeString)typeStr.wrapValue("Hello");
```

Is equivalent to:

```
NativeString natStr2 = (NativeString)typeStr.newString("Hello");
```

Both native capable string objects contain the Java String `"Hello"`. When any of them is made native, the `"Hello"` string using `wchar_t` characters and ended with `'\0'`, will be written on the related native memory associated.

5.1.1.3 Example: an array of Unicode string pointers.

```
VarTypeNative varTypeStr = typeStr.decVarType();

TypeNativeArray typeStrArray = typeMgr.decArray(2,varTypeStr);
```

The returned type specifies string arrays with 2 elements.

```
NativeStringArray strArray = (NativeStringArray)typeStrArray.wrapValue(

            new String[]{"hello","bye"});
```

When this object is made native, two new native strings are written to the native memory with the Unicode strings `"hello"` and `"bye"` and the two addresses are written on the new native array with two string pointers (`wchar_t*`).

5.1.2 Creation with factories

The interface factory, `NativeCapableFactory` , may be used as a shortcut to create objects of predefined native capable classes.

5.1.2.1 Example 1

```
NativeCapableFactory objFact = JNIEasy.get().getNativeCapableFactory();

NativeString natStr = objFact.newString("Hello");
```

Creates a `NativeString` object with the default encoding (usually ANSI)

5.1.2.2 Example 2

```
NativeString natStr2 = (NativeString)objFact.wrapValue("Hello");
```

Is equivalent to the previous example, the method `NativeCapableFactory.wrapValue(Object)` expects a "can be native" object to be wrapped with the matching native capable object, the native type used is the default associated to the "can be native" object (an ANSI string).

5.1.2.3 Example 3

```
NativeStringArray strArray = (NativeStringArray)objFact.wrapValue(

                new String[]{"hello","bye"});
```

Creates a native capable string array holding the specified string array, the default string encoding will be used when made native.

5.1.2.4 Example 4

```
NativeInteger natInt = objFact.newNativeInteger(10);
```

Creates a `NativeInteger` object holding the value 10.

5.1.3 Creation of proxies of DLL methods and fields

The proxies of DLL methods (and fields seen as methods) are usually created using the `DynamicLibrary` instance of the DLL.

The following example maps and calls the C Win32 method:

```
HWND FindWindowA(LPCTSTR, LPCTSTR)
```

located in the Windows `User32.dll`, from Java:

```
DLLManager dllMgr = JNIEasy.get().getDLLManager();

DynamicLibrary dll = dllMgr.get("User32");

CMethod method = dll.addCMethod("FindWindowA",int.class,

            new Object[]{String.class,String.class},CallConv.STD_CALL);

int hwnd = method.callInt(new Object[]{null,"DDE Server Window"});
```

The returned `CMethod` object is already native and maps the specified C method, the `callInt` method execute a native call from Java. Note the `.dll` extension is removed in "`User32`" DLL name, the `DLLManager.get(String)` method can follows the `System.loadLibrary(String)` conventions: no extension and "lib" prefix (in Unixes) must be used, unless an absolute path is used.

JNIEasy uses `java.library.path` system property to specify where to look for dynamic libraries/shared objects (unless an absolute path is specified). This system property is usually set with the parameter `-Djava.library.path` on the command line.

Alternatively JNIEasy allows to specify `java.library.path` on runtime with the call:

```
JNIEasy.setFeature("java.library.path","pathList");
```

This call doesn't change the real `java.library.path` system property (mandates JNIEasy to ignore the system property and use the specified path list).

5.1.3.1 Field-methods

C global exported variables or C++ exported static fields can be accessed as native field-methods; there is no need to exist a "real" native field-method in the DLL.

The following example declares a global C variable exported in `MyLibrary.dll/libMyLibrary.so/libMyLibrary.dylib`:

```
// Any C or C++ .h file

extern "C" DLLEXPORT int aGlobalVar;
```

```
// Any C or C++ .c/.cpp file

int aGlobalVar;
```

Where `DLLEXPORT` macro is defined in `JNIEasy.h` as `__declspec(dllexport)` in Win32 compilers (VisualC++, MinGW, cygwin) and none in gcc Linux, Mac and Solaris (by default gcc on unixes exports all symbols[4]).

This variable is mapped from Java with a field-method native object.

The Java side:

```
DLLManager dllMgr = JNIEasy.get().getDLLManager();

DynamicLibrary dll = dllMgr.get("MyLibrary");

CFieldMethod fieldMethod =

        dll.addCFieldMethod("aGlobalVar",int.class,CallConv.STD_CALL);

int value = fieldMethod.callInt(NativeFieldMethod.GET,0); //or .getInt();

value += 10;

fieldMethod.callInt(NativeFieldMethod.SET,value) // or .setInt(value);
```

5.1.4 Creation of proxies of native methods and fields if the address is known

Another way to create a proxy of a native method is using `NativeConstructor`, `NativeStaticMethod`, `NativeInstanceMethod`, `NativeStaticFieldMethod` and `NativeInstanceFieldMethod` interfaces. Instances implementing these interfaces may be created using a native capable method signature. The native capable signature can be created with the `NativeSignatureManager` utility object, this object is obtained with the method `JNIEasy.getSignatureManager()`.

In the following example the native method address is obtained with the method `DynamicLibrary.getAddress(String)`, another scenarios are possible (including fields of structures declared as pointer to method)

```
DynamicLibrary dll = JNIEasy.get().getDLLManager().get("User32");

long methodAddress = dll.getAddress("FindWindowA");

NativeSignatureManager sigMgr = JNIEasy.get().getSignatureManager();

NativeStaticMethodSignature sig = sigMgr.decStaticMethod(int.class,

            new Object[]{String.class,String.class},CallConv.STD_CALL);

NativeStaticMethod method =
```

[4] Inline methods (C and C++) and C static methods (using the `static` keyword) are not exported by default. A new visibility model have been introduced in gcc, more info: http://gcc.gnu.org/wiki/Visibility and http://people.redhat.com/drepper/dsohowto.pdf

```
(NativeStaticMethod)sig.attachBehavior(methodAddress);

int hwnd = method.callInt(new Object[]{null,"DDE Server Window"});
```

The returned `NativeStaticMethod` object is already native and maps the specified C method, the native call using the method `callInt`, invokes the native method in `User32.dll` from Java.

5.1.4.1 Field-Methods

A special case is field-methods, the native address must be, really, the address of a native field-method. The following address:

```
long variableAddress =

    dll.getAddress("aGlobalVariableOrStaticExportedFieldName");
```

is not a valid address to attach a `NativeStaticFieldMethod`, but the address of a `CFieldMethod` (obtained using `NativeCapableUtil.getAddress(Object)`) is a valid field-method address.

5.1.5 Creation of direct and reflection callbacks

A native capable object of these types may be created using a native method signature and a Java native capable method to call, this method (the method to be finally called) is represented by a reflection object (`Method`). A direct or reflection callback maps the Java method and can be passed to the native side (because they are native capable objects), the native side sees them as pointer-to-method, if this pointer-to-method is called the Java method is finally called, direct and reflection callbacks work as intermediaries in the calling process. A native object callback allocates native memory and has an address, this address is the native "pointer" of the Java method (of course is an illusion, it is the address of a native method created on the fly working as a bridge); this memory is freed when the object is garbage collected.

Example:

```
// Static method: java.lang.Math.max(int,int)

Method method = Math.class.getDeclaredMethod("max",

                        new Class[]{int.class,int.class});

NativeSignatureManager sigMgr = JNIEasy.get().getSignatureManager();

NativeMethodSignature sig =

                    sigMgr.decMethod(method,CallConv.STD_CALL);
```

Creation of a new reflection based callback and invoking it with a native call from Java:

```
NativeManager natMgr = JNIEasy.get().getNativeManager();

NativeStaticMethodReflection objReflect =

    (NativeStaticMethodReflection)sig.newMethodReflection(method);
```

```
natMgr.makeNative(objReflect);

int max = objReflect.callInt(new Object[]{new Integer(5), new Integer(3)});

System.out.println("Must be 5: " + max);
```

Creation of a new direct based callback and invoking it with a native call from Java:

```
NativeDirectStaticMethodCallback objDirectCb =

  (NativeDirectStaticMethodCallback)sig.newDirectMethodCallback(method);

natMgr.makeNative(objReflect);

int max2 = objDirectCb.callInt(new Object[]{new Integer(5),new Integer(3)});

System.out.println("Must be 5: " + max2);
```

The objects, `objReflect` and `objDirectCb`, wrap the specified Java method (`Math.max(int,int)`) and make it visible from native. The reflection object (`objReflect`) uses reflection to call the method and the direct object (`objDirectCb`) uses a normal Java call (this object is an instance of a generated Java class "on the fly" if necessary). In this example the Java method can be called from native with a pointer with the signature: `int (*)(int,int)`

The method `NativeManager.makeNative(Object)` has been used the first time, this method converts a native capable object to a native object, after the call the Java object is associated to a chunk of native memory, in this case the cited native method created on the fly working as a bridge.

Following the example:

```
long address = NativeCapableUtil.getAddress(objReflect);

NativeStaticMethod proxy =

                (NativeStaticMethod)sig.attachBehavior(address);

int max3 = proxy.callInt(new Object[]{new Integer(5),new Integer(3)});

System.out.println("Must be 5: " + max3);
```

Using these techniques, native Java callbacks can be tested using Java only.

5.1.5.1 Field-methods

Java native capable fields, static or instance, can be exported to the native world using direct and reflection native callbacks too. The fields do not need to be declared native, because reflection or direct access is used, in the case of instance fields, the container class must be native capable and a native instance must be used, but the field itself may be non-native (but the field type must be native enabled).

Example:

```
// Accessing the public final static field: java.lang.Math.PI

Field field = Math.class.getDeclaredField("PI");
```

```
NativeSignatureManager sigMgr = JNIEasy.get().getSignatureManager();
NativeFieldMethodSignature sig =
                        sigMgr.decFieldMethod(field,CallConv.STD_CALL);
NativeManager natMgr = JNIEasy.get().getNativeManager();


// Using reflection


NativeStaticFieldMethodReflection objReflect =
  (NativeStaticFieldMethodReflection)sig.newFieldMethodReflection(field);
natMgr.makeNative(objReflect);
double Pi1 = objReflect.getDouble();
System.out.println("Must be true: " + (Pi1 == java.lang.Math.PI));


// Using a direct access


NativeDirectStaticFieldCallback objDirectCb =
    (NativeDirectStaticFieldCallback)sig.newDirectFieldCallback(field);
natMgr.makeNative(objDirectCb);
double Pi2 = objDirectCb.getDouble();
System.out.println("Must be true: " + (Pi2 == java.lang.Math.PI));
```

Of course these native objects can be passed to the native side as pointer-to-methods following the field-method signature, the native side can access the Java fields calling these field-methods.

5.1.6 Using Java reflection as proxy of native methods

The `NativeStaticMethodReflection` interface is mainly used in Java callbacks invoked using reflection, the native instance holds a `Method` reference and this reflection object is used to call to the Java method. But there is an exception the native instance does not hold a Method reference, the internal reflection object is not defined, and we try to get something with a call to `NativeStaticMethodReflection.getMethod()…` in this case the framework is instructed to construct a new `java.lang.reflect.Method` object representing the native method with the address of the native instance, **the native method can be called using this reflection object**.

Example:

```
DynamicLibrary dll = JNIEasy.get().getDLLManager().get("User32");

long methodAddress = dll.getAddress("FindWindowA");

NativeSignatureManager sigMgr = JNIEasy.get().getSignatureManager();

NativeStaticMethodSignature sig = sigMgr.decStaticMethod(int.class,

            new Object[]{String.class,String.class},CallConv.STD_CALL);

TypeNativeStaticMethod type =

                  sig.decStaticMethod(NativeStaticMethodReflection.class);

NativeStaticMethodReflection obj =

                        (NativeStaticMethodReflection)type.newValue();

JNIEasy.get().getNativeManager().attach(obj,methodAddress);

Method method = obj.getMethod();

Integer hwnd = (Integer)method.invoke(null,

                              new Object[]{null,"DDE Server Window"});

System.out.println("Handle: " + hwnd.intValue());
```

This feature is transparent to the user reading a `java.lang.reflect.Method` field declared as a static method in a user defined Java class declared as a structure (C++ class or union), this field represents a "native pointer to method", if this field was not set (from Java) before a first read, the `Method` object read represents the native method in the DLL.

If using the `NativeInstanceMethodReflection` interface in a similar scenario, the `Method` object returned with `NativeInstanceMethodReflection.getMethod()` is static, the C++ instance method can be called using as first parameter the native instance.

The `NativeConstructorReflection` interface works as expected; the `java.lang.reflect.Constructor` can be used to call the C method declared as constructor.

The interfaces `NativeInstanceFieldMethodReflection` and `NativeStaticFieldMethodReflection` do not work as proxy and an exception is thrown, the `java.lang.reflect.Field` can not be used as a proxy of a field-method.

5.2 CREATION OF USER DEFINED JAVA NATIVE CAPABLE OBJECTS

An instance of a user defined Java native capable class (previously enhanced) works like an instance of the non-enhanced version, until the instance is made native. Instances can be created with the Java `new` sentence using the selected constructor by the user, this instance is not native (is a native capable object).

Example:

```
MyStructure struc = new MyStructure();
```

User defined Java classes will be explained later.

6. RUNTIME LIFE CYCLE

The runtime life cycle of a native capable object is very simple, there are only two states: native and non-native.

Every native capable class has, at least, two protected transient instance fields defined by the framework (added by the enhancer if the class is user defined). If the class inherits from another native capable class, both fields are present in the top most native capable class:

- `TypeNative jnieasyType`

  References the native type declaration to be used by the object when native. The referenced native type object is ever present although the object is not native. It can be obtained with the method `NativeCapable.jnieasyGetType()`.

- `NativeStateManager jnieasyNativeStateManager`

  References the native state manager object. This per-instance object manages the native state of the native instance. It can be obtained with the method `NativeCapable.jnieasyGetNativeStateManager()` or with `NativeCapableUtil.getNativeStateManager(Object)`

A native capable object is native when a `NativeStateManager` object is associated, the `jnieasyGetNativeStateManager()` method or returns this object , if not native returns `null`.

6.1 FRAMEWORK INITIALIZATION

First of all the framework must be loaded/initialized:

```
JNIEasy jniEasy = JNIEasy.get();

jniEasy.load();
```

This sentence loads the JNIEasy's dynamic library/shared object (`JNIEasy.dll` on Windows, `libJNIEasy.so` on Linux and Solaris, `libJNIEasy.jnilib` on Mac OS X).

On loading, the framework needs to locate the product's license file, `JNIEasy.lic`, this file is looked for in the directory specified with the optional `JNIEASY_LICENSE_DIR` Java property (specified with a –D flag on command line), otherwise the current directory is looked for.

Optionally the license directory can be specified programmatically using the method `JNIEasy.setFeature(String,Object)` and the feature "`jnieasy.license.dir`" **before** calling the `load()` method:

```
JNIEasy jniEasy = JNIEasy.get();

jniEasy.setFeature("jnieasy.license.dir","some path");

jniEasy.load();
```

6.2 MAKING NATIVE A NATIVE CAPABLE OBJECT

There are several ways to make native a native capable object:

6.2.1 Allocating native memory

The method `NativeManager.makeNative(Object)` makes native the native capable object specified. This call allocates the necessary native memory, creates a new `NativeStateManager` object to manage this memory, associates to the object, and "unfetches" the values of contained native fields to the native memory recursively (a "deep" unfetch). After this call the object and referenced by fields (reachable) native capable objects are made native and the allocated memory is in sync with the field values before the call.

The primitive fields are simply copied to the matched native memory.

An object referenced by a native field with a native capable class type, is made native:

1. Allocating native memory if the field is declared "by pointer": the address of the new memory block is set to a pointer of the container's memory according to the native memory layout.

2. Or attaching to the start of a memory part of the container's native memory if the field is declared "by value", according to the native memory layout.

3. If the native field type is a "can be native class" (e.g. `String`, `int[]`) an internal auxiliary native object is used holding the object referenced by the field. This auxiliary object is made native as explained before.

4. The native instance owns its related native memory; the pair native instance-address is registered internally by the framework.

Only one object can be the owner of a native address.

A shortcut to obtain the native address is invoking the static method `NativeCapableUtil.getAddress(Object)`; the same instance is returned by the method `NativeManager.findObject(long)` using the previously returned address (it only works if the object owns the memory associated).

If the framework needs a native object to represent a native address, first looks in the internal registry to locate the native object owning this address, if not found creates a new native capable object (with the expected type) and attaches it to the address.

Example:

Consider this class declared as a C structure in the XML enhancer descriptor of the class (only fields are declared as native, no methods):

```
package examples.manual;

import com.innowhere.jnieasy.core.*;

import com.innowhere.jnieasy.core.data.*;

import com.innowhere.jnieasy.core.factory.*;

import com.innowhere.jnieasy.core.mem.*;
```

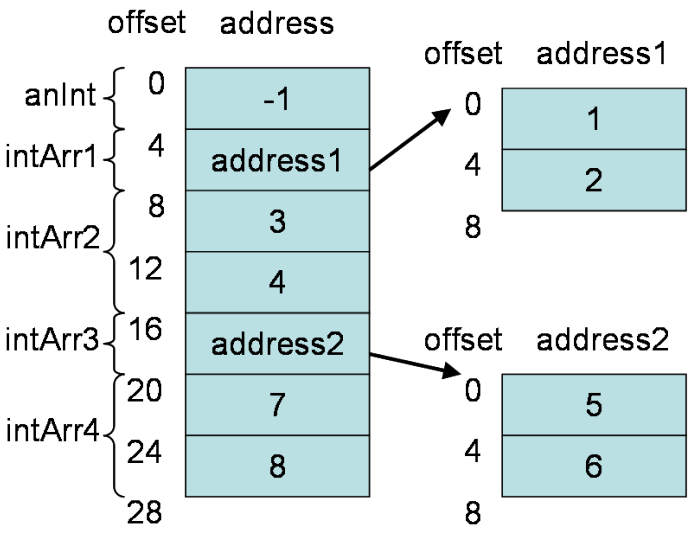
```
public class MyStructure
{
    protected int anInt = -1;
    protected NativeIntegerArray intArr1;
                        // Declared "by pointer" length undefined
    protected NativeIntegerArray intArr2;
                        // Declared "by value" length 2
    protected int[] intArr3 = new int[]{5,6};
                        // Declared "by pointer" length 2
    protected int[] intArr4 = new int[]{7,8};
                        // Declared "by value" length 2

    public MyStructure()
    {
        NativeCapableFactory factory =
                            JNIEasy.get().getNativeCapableFactory();
        intArr1 = (NativeIntegerArray)factory.wrapValue(new int[]{1,2});
        intArr2 = (NativeIntegerArray)factory.wrapValue(new int[]{3,4});
    }
}
```

An example of use:

```
MyStructure obj = new MyStructure();
NativeManager natMgr = JNIEasy.get().getNativeManager();
natMgr.makeNative(obj);
long address = NativeCapableUtil.getAddress(obj);
Object obj2 = natMgr.findObject(address);
System.out.println("Must be true: " + (obj == obj2));
```

Supposed a 32 bit platform, the native memory layout after made native is:



6.2.2 Attaching native memory

A native capable object may be attached to a previously known (supposed valid) native address and memory zone starting in this address, corresponding to the expected memory layout of the instance. When attaching, a new `NativeStateManager` object is created, attached to the native address and associated to the object (now native).

The attached native instance does not own its related native memory. **Several native objects may be attached to the same native address** and they are not registered as "owners" of the address. A native instance may be attached to an address owned by another native instance (created with `NativeManager.makeNative(Object)`); the framework tries to avoid this because it looks first for a native instance owning the address, but the user can attach explicitly.

There are two ways to attach a native capable object to a native address:

6.2.2.1 Native memory is overwritten (unfetch-attach)

The method `NativeManager.makeNative(Object,long)` is used. The current values of the native fields are mandatory over the native memory, and native memory is overwritten ("unfetched") following the rules explained before in the Allocating native memory section using the method `NativeManager.makeNative(Object)`. The differences are: no new native memory is allocated and the attached native object does not own the associated native memory/address.

Example:

```
MyStructure original = new MyStructure();

NativeManager natMgr = JNIEasy.get().getNativeManager();

natMgr.makeNative(original);

original.setIntArray3(null);

System.out.println("Must be null: " + original.getIntArray3());

long address = NativeCapableUtil.getAddress(original);
```

```
MyStructure attached = new MyStructure();

natMgr.makeNative(attached,address);

attached.setIntArray3(new int[]{1,2});

/* Modifies the native memory of "original" */

System.out.println("Must be 1: " + original.getIntArray3()[0]);
```

6.2.2.2 Native memory is kept untouched (attach)

The method `NativeManager.attach(Object,long)` is used. The native memory state is mandatory over the fields, but there is no fetch of the native memory to fields, because when a Java field is read, the field is synchronized before with the native value (the Java value is set with the native value, native memory is fetched). Using this lazy synchronization increases the performance if a small part or no fields are used and ensures the field value read is ever the current native memory value: native memory can be modified from native in any time, a very important feature in a multithread programming world. Of course any Java field updated writes the new value to the native memory (there is no difference between an object attached or owning the native memory when fields are read and written). To ensure the Java fields are fully synchronized with the native memory, the method `NativeManager.fetch(Object,long)` can be called after the attach call.

Example:

```
MyStructure original = new MyStructure();

NativeManager natMgr = JNIEasy.get().getNativeManager();

natMgr.makeNative(original);

original.setIntArray3(new int[]{1,2});

System.out.println("Must be 2: " + original.getIntArray3()[1]);

long address = NativeCapableUtil.getAddress(original);


MyStructure attached = new MyStructure();

natMgr.attach(attached,address);

System.out.println("Must be 2: " + attached.getIntArray3()[1]);
```

6.2.3 By reachability of the fields

In the previous cases the referenced native capable objects of native fields are made native by reachability when the container object is made native or when is first accessed if attached (without unfetch).

Following the previous example:

```
public class MyStructure
{
...
    public NativeIntegerArray getIntArray1()
    {
        return intArr1;
    }

    public void setIntArray1(NativeIntegerArray intArr1)
    {
        this.intArr1 = intArr1;
    }
}

MyStructure obj = new MyStructure();
NativeManager natMgr = JNIEasy.get().getNativeManager();
natMgr.makeNative(obj);
NativeCapableFactory factory = JNIEasy.get().getNativeCapableFactory();
NativeIntegerArray newValue =
                (NativeIntegerArray)factory.wrapValue(new int[]{10,11});
obj.setIntArray1(newValue);
```

The new value, `newValue`, is not native before updating the `intArr1` field; the new referenced object will be made native automatically before the field is set, because the field is native and the method code is managed by the enhancer.

6.3 READING AND UPDATING NATIVE INSTANCES

The native fields of user defined native capable classes must not be accessed directly outside the class, because any access must be done on managed (enhanced) code to be policed by the enhancer, currently only the instance member methods and constructors of the native capable class are enhanced (managed code), any access of the native fields is monitored and synchronized, if necessary, with the native memory.

The fields of predefined classes are ever accessed using the implemented methods of the related native interface; these methods synchronize the fields with the native memory as the enhancer does with user defined classes.

6.3.1 Reading fields of user defined native capable classes

The native memory is ever mandatory over the current values of the fields. When a native field is read, the native memory is **ever** read first, and the field value is modified with the current matched value of the native memory, although the field has not been modified after the last read on the Java side. Any reading action must be seen as "I want to read the native memory", because the "Java state" may remain the same between readings, but the native memory may have changed in the native side, or with direct access to the native memory using the `NativeBuffer`, into the same or another thread (the threads synchronization is a developer's work).

Although the field is read from the user point of view, the field value is updated if native memory is different to the current field value.

The following descriptions apply to fields of user defined native capable classes. The default fetch mode is supposed to be "fast".

6.3.1.1 Reading a field with a primitive type

The native memory is read and copied to the field and returned.

6.3.1.2 Reading a field with a native capable class type declared "by pointer"

The matched native memory of the field is a pointer.

a) If the address value is 0 (`null`): the field is set to `null` and the current native object referenced, if any, is lost (this object remains unchanged, if not referenced any more is garbage collected).

b) If the address value is not `null` : the address is compared with the address of the current referenced native object (if any), if equal the current referenced object is returned, if not equal, the framework search in the internal registry the native instance owning the address, if found this object is returned (the object pointed) and the current object is discarded, if not found and the current object is native then is discarded (pointer has changed) and a new attached object to the address is returned, else (if not native) the current object is attached to the address and returned.

6.3.1.3 Reading a field with a native capable class type declared "by value"

The matched native memory is a part of the container object's memory, the address is the start of this memory part (is managed by the framework as the pair container's address - relative offset of the field). The field needs to reference a native object attached to this address (managing this memory part), if the field is `null` a new object is automatically created with the expected type, attached and returned. If the current referenced object already exists and is already native (then attached to the matched memory part), it is returned as is, and no synchronization of the instance and the native memory is performed (if the default fetch mode is set to "fast"), because any access to the fields of the returned object will perform this synchronization.

In brief: the native object attached to the matched native memory part is returned, and is previously created and attached if necessary.

6.3.1.4 Reading a field with a "can be native" class type declared "by pointer"

An auxiliary native object linked to the field, implementing the `CanBeNativeCapable` interface, is used; this auxiliary object is managed by the container's `NativeStateManager` object. This auxiliary object is managed following the same rules of a field with a native capable class "by pointer", as if the field type was a native capable class referencing this auxiliary object. This object is not finally returned because is expected a "can be native" object (a `String`, `int[]` etc.), the auxiliary object, if not `null`, is used to obtain the "can be native" object with the same value of the native memory "pointed" calling the method `NativeSingleFieldContainer.getValue(int)` using the default fetch mode ("fast" by default).

6.3.1.5 Reading a field with a "can be native" class type declared "by value"

In this scenario the field must be not `null`, because the referenced object represents the always present related memory part of the container's native memory (when native of course), otherwise an exception is thrown.

When reading, an auxiliary object is needed too, as the "by pointer" case, but this object is attached to the matched native memory part represented by the field, following the same rules of a field with a native capable class "by value", and never is discarded. Because the current "can be native" object is never `null`, the auxiliary object used to obtain the expected "can be native" object is never `null` too (is created if necessary), and again the "can be native" object to return is obtained with the method `NativeSingleFieldContainer.getValue(int)` using the default fetch mode ("fast" by default), the returned "can be native object" is previously synchronized with the matched native memory part; if the "can be native" object is modifiable (`int[]` etc) the current but modified object is returned, else a new object is created, synchronized and returned.

6.3.2 Updating fields of user defined native capable classes

Again the framework supposes the native memory may be update in the native side, or using direct access to the native memory using the `NativeBuffer`. When a native field is updated, the native memory is **ever** updated too although the new value is the same as the current value of the field, this ensures that the native memory is really updated. Any updating action must be seen as "I want to update the native memory", because the "Java state" may remain the same but the native memory may have changed indirectly. A valid technique is to update a field with the same value to ensure the native memory is set with this value.

The following descriptions apply to fields of user defined native capable classes. The default "unfetch" mode is supposed to be "fast".

6.3.2.1 Updating a field with a primitive type

The field and related native memory are updated with the new value.

6.3.2.2 Updating a field with a native capable class type declared "by pointer"

The matched native memory is a pointer. If the new object value is `null` the pointer is set to 0, else the new object is made native, if necessary, and its address is set as the value of the pointer in the native memory. Any previously referenced object by the field is unreferenced.

6.3.2.3 Updating a field with a native capable class type declared "by value"

The matched native memory is a part of the container's memory. The new object can not be `null` (otherwise an exception is thrown), if the object is not native is made native with an unfecth-attach otherwise the native memory is not modified (if the default "unfetch" mode is "fast").

Updating a "by value" field must be used to update the container object managing the matched native memory part, but not the native memory itself, because the field value (the manager of the memory part) may be used itself to update the native memory, using its methods or passing as a parameter to `NativeManager.unFetch(Object)`.

6.3.2.4 Updating a field with a "can be native" class type declared "by pointer"

If the new value is `null` the matched native pointer is set to 0 and the current auxiliary object is discarded, else if the auxiliary native object is missing, a new object is created and made native wrapping the new "can be native" value (the native memory of the new object is updated with the "can be native" value), else if the new value is not equal to the current field value, the current auxiliary object is discarded and a new auxiliary object is created and made native wrapping the new value. Unless the new value is `null`, the native address of the final auxiliary object is used to update the native pointer, else is set to 0. The native memory is ever updated although the new value is the same as current value because the native pointer may be changed or in the native side, or using direct access to the native memory.

6.3.2.5 Updating a field with a "can be native" class type declared "by value"

The auxiliary object is used, it is created and unfetch-attached if was not; this object is used as the manager of the matched native memory part and never is discarded as said in "reading". The new "can be native" value is used to update the native memory part using the auxiliary object with the default "unfetch" mode.

Note that the native memory is ever updated although the new value is identical to the current value because the native memory may have been updated indirectly.

6.3.2.6 Updating elements of Java array fields

The enhancer does not control the updating of an element of a Java array field.

Considering the following code:

```
public class MyStructure

{

...

    protected int[] intArr3 = new int[]{5,6};

                                    // Declared "by pointer" length 2

...

    public int[] getIntArray3()

    {

        return this.intArr3;
```

```
    }

    public void setIntArray3(int[] intArr3)
    {
        this.intArr3 = intArr3;
    }
    public void setIntArray3(int index,int value)
    {
        this.intArr3[index] = value;
    }
}
```

The fields access "`this.intArr3[index]`" implies a native memory read updating the current Java array value with the native memory. But the element modification is not detected by the enhancer and no native memory update is performed.

Several solutions to this problem:

1. Updating the method as:

```
public void setIntArray3(int index,int value)
{
    int[] aux = this.intArr3;
    aux[index] = value;
    this.intArr3 = aux;
}
```

   The last sentence ensures the native memory is updated with the whole array.

2. Re-setting the modified array:

```
MyStructure obj = ... // a native instance
int[] arr = obj.getIntArray3();
arr[index] = ... // element modifications
obj.setIntArray3(arr); // updates the whole array
```

3. Using a corresponding native class array wrapper as field and use "by element access" methods:

```
protected NativeIntegerArray intArr3;
```

The `NativeIntegerArray` interface has the method `void setInt(int index,int value)` updating only the selected element in native memory. The method `int getInt(int index)` is more performant too because only the selected element is read from native memory.

This alternative increments the performance and control but decreases the transparency, is only recommended with big arrays used with frequent access to the elements.

6.3.2.7 Updating elements of `StringBuffer` fields

The enhancer does not control the updating of a `StringBuffer` using its methods, use similar techniques seen with Java arrays.

6.3.3 Reading/writing the native memory of `CanBeNativeCapable` objects

The `CanBeNativeCapable` objects enable the developer to read from and write to native memory "can be native" objects. These utility objects are used as auxiliary objects to manage "can be native" fields behind the scenes; these objects can be used explicitly too.

6.3.3.1 Reading/Writing `NativeString` based objects

When a `NativeString` object is made native allocating memory, the current `String` value is copied (using the specified encoding) in the native memory, the memory size is the string length + 1 (the `'\0'` character) multiplied with the character size, this memory size can not change. If the object is attached and no `String` was set before, in this case the memory size is undefined, this case is specially useful on reading strings with unknown length (the `NativeString` object is attached to native memory and the length is unknown), usually using the method `NativeString.getString()`.

When reading, the current `String` object is returned if the native memory contains the same string as the current `String` object, else a new `String` is returned.

When writing, usually using the method `NativeString.setString(String)`, the new value is written in the native memory, but the string length can not be greater than the length of the original `String` value immediately before the container is made native allocating memory, unless the object was attached and no `String` was set before (memory size is undefined), in this case the framework can not check an out of bounds writing.

6.3.3.2 Reading/Writing `NativeStringBuffer` based objects

The behavior is mainly the same as `NativeString` objects, with only one difference: when reading, the returned `StringBuffer` is ever the current object because this object is modifiable.

With the methods `getCharacter(int index)` and `setCharacter(int index,int value)`, individual characters can be read/modified, this supposes a performance gain instead of reading/updating the complete string. Furthermore using these methods can be read/updated individual characters of very big native strings without duplication in Java: if a `NativeStringBuffer` object is attached to a known address and no `StringBuffer` was set before, the memory size is unknown and no internal `StringBuffer` object was created (no duplication of the string), the methods `getCharacter(…)` and `setCharacter(…)` can be used

to access individual characters without having read/update the whole native string; if the method `getStringBuffer()` is used the whole native string is read (supposed that a final `'\0'` exists) and duplicated inside an internal `StringBuffer` object.

6.3.3.3 Reading/writing arrays with undefined length

In a similar fashion as `NativeStringBuffer` objects, any native object implementing the `NativeArray` interface can deal with undefined length arrays: if an object is attached to a known address and no Java array was set before, the memory size is unknown and no internal Java array is hold, and can not because array length is unknown, the methods like `NativeObjectArray.getObject(int index)`, `NativeObjectArray.setObject(int index,Object value)`, `NativeIntegerArray.getInt(int index)`, `NativeIntegerArray.setInt(int index)` and so on, can be used to read/update individual elements without read/update the entire array.

6.3.3.4 Reading/writing primitive arrays

These objects implement the interface `NativePrimitiveArray`. Primitive arrays are optimized to read and write big arrays, because the copy operation between native memory and Java array (both directions) is realized using the complete memory block of the array. Remainder array classes read and write element by element.

6.3.3.5 Reading/writing arrays of "can be native" objects

These objects implement the interface `CanBeNativeCapableArray`, a native instance uses internally an auxiliary array (with the same length) of auxiliary native objects. The relation of the pair array element-auxiliary native objects follow the same relationship explained on "can be native" fields of user defined classes; the processing is different if the array holds elements "by value" of "by pointer".

With undefined length arrays the auxiliary array dynamically increase until necessary to manage the selected element read/updated.

6.3.4 Fetch and "unfetch" modes

"Fetch" is the operation to read from native memory and copy to Java memory. "Unfetch" is the operation to write the native memory with the data copied from Java memory.

All reading and writing operations between Java and native memory have any kind of "deepness". There are four types of deepness:

1. NONE: there is no copy between Java memory and native memory, the reading and writing operations are performed on Java memory only. The NONE mode may be used to read or update Java fields with no native memory interaction.

2. FAST: only field members with primitive types, "can be native" class types and the address of fields with native capable class types "by pointer", are synchronized with the native memory; fields with native capable class types "by value" are not synchronized because the native "embedded" object can be used, itself, to access the associated native memory. The referenced native object of a field with a native capable class type declared "by pointer", is not synchronized itself, only the object's address and the related pointer in the container's native memory are synchronized.

3. EMBEDDED: the native objects referenced by fields with native capable class types declared "by value" (native embedded objects) are synchronized, themselves, with the related native memory too.

4. DEEP: the native objects referenced by fields with native capable class types declared "by pointer" (native linked objects) are synchronized, themselves, with the related native memory too. All members of the native object are synchronized recursively. Cyclic references are detected with no problem.

The following table shows if a field is synchronized (fetched/unfetched) with the native memory if accessed (read/write) depending on the fetch/unfetch mode and field type:

Native Field Type	NONE	FAST	EMBEDDED	DEEP
Primitive	NO	YES	YES	YES
"Can be native" (by value & by pointer)	NO	YES	YES	YES
Native capable by pointer	NO	YES, address only	YES, address only	YES, referenced object's fields with DEEP mode too
Native capable by value (referenced object already attached)	NO	NO	YES, the referenced object's fields with EMBEDDED mode too	YES, the referenced object's fields with DEEP mode too

The default mode to fetch and "unfetch" is FAST, the fast mode ensures a reasonable synchronization of Java memory and native memory.

The `Fetch` y `UnFetch` interfaces define the constants used to specify the fetch and "unfetch" modes.

The developer can:

a) change the default fetch and "unfetch" mode of every read and write operation with the methods `NativeManager.setDefaultFetchMode(int)` and `NativeManager.setDefaultUnFetchMode(int)`

b) use the `fetchMode` or `unFetchMode` parameters present in multiple read/write methods of `NativeCapable` inherited interfaces

c) execute a fetch or "unfetch" operation over a native instance with the methods `NativeManager.fetch(Object obj,int mode)` and `NativeManager.unFetch(Object obj,int mode)`.

6.4 FREEING A NATIVE INSTANCE

If the native instance owns its native memory (was made native with `NativeManager.makeNative(Object)`); this native memory is automatically freed if the native instance goes out the scope and is reclaimed by the garbage collector. This is realized by the `finalize()` method of the internal implementation of `NativeStateManager`, the `finalize`

method of the native instance is not used and the developer can use it with freedom in user defined classes.

There are two methods to free the native memory of a native object explicitly:

```
NativeManager.free(Object)

NativeCapableUtil.free(Object)
```

Both methods are roughly equivalent: if the object owns its native memory, the native memory is freed else does nothing, in both cases the native instance is made not native. The embedded native object fields are freed too (native objects pointed by fields "by pointer" are untouched).

There are two methods to detach a native object from the native memory without freeing it:

```
NativeManager.detach(Object)

NativeCapableUtil.detach(Object)
```

Both methods are roughly equivalent too: the object is detached from the related native memory and made not native. The embedded native object fields are detached too (native objects pointed by fields "by pointer" are untouched). The developer must free the detached native memory if necessary (taking care with memory leaks).

To free (and make not native) an entire object tree (freeing linked objects too) the following method can be used:

```
NativeManager.detach(Object obj, int freeMemMode, boolean deep)
```

with the parameters `freeMemMode` set to `FREE_MEMORY` and `deep` to `true`.

7. DEFINITION OF USER DEFINED JAVA NATIVE METHODS AND CLASSES

The developer can define several types of user defined classes mapping native constructions:

- C++ classes
- Structures (C and C++ based)
- Unions (C and C++ based)
- Wrappers of native capable arrays
- Wrappers of multidimensional arrays (arrays of arrays)
- Pointer types (wrappers of pointers)
- Direct callbacks

Any other native construction is solved with a predefined class of the framework, in fact, the user defined array wrappers and pointer classes are not needed at all because generic predefined classes are used by the framework to wrap the contained elements, but user defined classes of these types enable the developer to avoid castings mainly.

The user defined classes must be enhanced to be used by the framework using a XML descriptor (describing the "native view" of the class); the enhancement converts them in native capable classes. This enhancement can be made before using in runtime or "on class loading", in this last case the XML descriptor must be present alongside the .class file.

7.1 C++ CLASSES, STRUCTURES, UNIONS AND C METHODS

C++ classes, structures and unions are basically the same in C++ and in JNIEasy's Java classes: they can have fields, constructors, methods, and inheritance (except unions). C methods or C++ static methods are basically the same: they can be mapped with Java methods inside (of course) Java classes.

In JNIEasy a user defined native capable class is a normal Java class, a POJO; this class can have declared native elements: fields, methods and a base class. A declared native element must use native enabled types and must be declared in XML as native (or declared by default as native like fields), elements not declared as native are not managed by the framework and do not have a "native correspondence". Example: a non-native field is not reflected in native memory, a non-native method is not a proxy of a DLL method or exported to the native world as a native callback.

7.1.1 Declaring native fields

The `MyStructure` class seen before is an example of a simple C/C++ structure in Java:

```
package examples.manual;
```

```
import com.innowhere.jnieasy.core.*;

import com.innowhere.jnieasy.core.data.*;

import com.innowhere.jnieasy.core.factory.*;

import com.innowhere.jnieasy.core.mem.*;


public class MyStructure

{

    protected int anInt = -1;

    protected NativeIntegerArray intArr1;

                            // Declared "by pointer" length undefined

    protected NativeIntegerArray intArr2;

                            // Declared "by value" length 2

    protected int[] intArr3 = new int[]{5,6};

                            // Declared "by pointer" length 2

    protected int[] intArr4 = new int[]{7,8};

                            // Declared "by value" length 2


    public MyStructure()

    {

        NativeCapableFactory factory =

                            JNIEasy.get().getNativeCapableFactory();

        intArr1 = (NativeIntegerArray)factory.wrapValue(new int[]{1,2});

        intArr2 = (NativeIntegerArray)factory.wrapValue(new int[]{3,4});

    }

...

}
```

The XML enhancer descriptor may be:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!-- Archive MyStructure.jnieasy.enh.xml -->
<jniEasyEnhancer version="1.1">
    <package name="examples.manual">
        <imports />
        <class name="MyStructure" type="structure">
            <field name="intArr2" varConv="byValue" length="2" />
            <field name="intArr3" length="2" />
            <field name="intArr4" varConv="byValue" length="2" />
        </class>
    </package>
</jniEasyEnhancer>
```

Note the absence of declarations of `anInt` and `intArr1` fields, in a XML enhancer descriptor all instance fields are considered as native by default (unless explicitly excluded or qualified with the `transient` keyword), and default configurations are valid to these fields: primitives do not need special native information, and object based fields are declared "`byPointer`" by default and array fields "`byPointer`" have undefined length by default.

The code of instance methods is ever enhanced and managed by the framework to detect any use of native declared instance fields (to synchronize with the native memory). If the following method was added to `MyStructure`:

```
public void addToAnInt(int value)
{
    this.anInt = this.anInt + value;
}
```

This method increments the related native memory with `value` without dependence of the current value of `anInt` field, because inside "`= this.anInt +`" part, the `this.anInt` access makes a transparent read of the native memory and updates the field before the addition, then its value is added to `value` and finally the temporal result is set to the `anInt` field ("`this.anInt = …`") updating the native memory too. The "`this.`" qualification is optional in this example (used to enforce the idea of using the field).

The previous method body is equivalent (and works the same), to:

```
this.anInt += value;
```

Because in the bytecode level there is no difference (the += operator is expanded).

Out of the enhanced Java class the fields are accessed without management/interception and native memory is not involved; **encapsulation of any external access to the fields with**

get/set methods is strongly encouraged to encapsulate any external access to the fields with get/set methods.

7.1.1.1 How select the appropriate native enabled type of fields starting with a real C/C++ native class/structure/union

In this manual, most of the time, "real" C/C++ constructions and types and ignored, because any valid JNIEasy construction, usually using Java primitive types, `String`, `StringBuffer`, arrays of these types, and user defined classes with methods and fields using these types, has a corresponding C/C++ construction and the opposite. Because the Java centric nature of JNIEasy, the developer must select the Java type matching the C/C++ type using the JNI definitions of the target platform. Example: in a 32 bit platform the `jint` type is the C `int` type, `jint` of course matches ever with the Java `int` type, the C `char` type matches with the Java `byte` type, a C `int[]` array (or `int*` pointing an array) matches with Java `int[]` and so on[5]. The developer has the option to use `NativeIntegerArray` instead of `int[]` , for instance, to take more control of the native management, `int[]` and `NativeIntegerArray` are interchangeable, same is applied to any other pair of "can be native" class and native capable wrapper.

In a 32 bit platform the matched C++ structure and initialization of `MyStructure` would be:

```
// MyStructure.h

struct MyStructure

{

    int anInt;

    int* intArr1;

    int intArr2[2];

    int* intArr3;

    int intArr4[2];


    MyStructure()

    {

        anInt = -1;

        intArr1 = new int[2];

        intArr1[0] = 1; intArr1[1] = 2;

        intArr2[0] = 3; intArr2[1] = 4;

        intArr3 = new int[2];
```

[5] This is the default native layout of Java primitives (and derivatives), but this correspondence can be changed, for instance: the Java `char` (2 bytes) can be reflected as a C `char` (1 byte), the Java `long` (8 bytes) can be reflected as a 32 bit C `int` etc. See Cross-platform options.

```
            intArr3[0] = 5; intArr3[1] = 6;

            intArr4[0] = 7; intArr4[1] = 8;

        }

    };
```

A C/C++ program accessing a `MyStructure` C/C++ structure with a `MyStructure*` pointer could access the native memory created with a native instance of the Java version or an object created on the native side could be accessed from Java attaching a `MyStructure` Java object to the address.

JNIEasy supports the "by value" convention to pass as a method parameter a structure, class, union or array (the last is not supported by C/C++ but may be useful to the developer in special cases). When using the "by value" convention the native memory of the formal parameter (the receiver) is a copy of the original argument value; in the structure, class or union cases the constructors of the copy are not invoked and virtual methods may be not valid, but non-virtual methods and fields can be accessed.

7.1.1.2 Memory alignment

The memory alignment of structures/classes/unions is not an issue unless specific alignment rules are used in C/C++ declarations (using `#pragma pack` or `__declspec(align(N))` in MSVC[6], and `__attribute__((aligned(N)))` in gcc).

To know the alignment of a C/C++ data type, the expressions `__alignof__(type)` in gcc and `__alignof(type)` in MSVC, can be used.

To know how a C++ instance field is aligned use the following macro:

```
#if defined(_MSC_VER) && (_MSC_VER >= 1300) && defined(__cplusplus)

# define FIELD_ALIGNMENT(type, field) __alignof(((type*)0)->field)

#elif defined(__GNUC__)

# define FIELD_ALIGNMENT(type, field) __alignof__(((type*)0)->field)

#else

/* FIXME: Not sure if is possible to do without compiler extension */

#endif
```

Several methods can be used to obtain alignment info of JNIEasy's native elements:

- `NativeTypeManager.alignSizeOf(Class)` : returns the preferred alignment size declared in the specified native enabled class.
- `TypeNative.preferredAlignSize()` : returns the preferred alignment size of the native type.
- `NativeClassDescriptor.alignSize()` : returns the effective alignment size of a structure type.

[6] Microsoft (Visual) C/C++ compiler

- `NativeFieldStructureDescriptor.alignSize()` : returns the effective alignment size of a concrete structure (class, union) field.

JNIEasy has four levels to control the memory alignment:

1. Global alignment specification

   The method `NativeTypeManager.setStructureAlignSize(long)` can be used to set the default memory alignment of structures, classes and unions; this value is imposed if lower than the preferred alignment of any field.

   The method `NativeTypeManager.getStructureAlignSize()` can be used to get the current default alignment size. The default value is 8 bytes.

2. Per structure alignment

   Using the optional XML enhancer attribute `alignSize` in the class declaration.

   Example:

   ```
   <class name="MyStructure2" type="structure" alignSize="4"> ...
   ```

3. Per field alignment

   Using the optional XML enhancer attribute `alignSize` in the field declaration.

   Example:

   ```
   <field name="someField" alignSize="1"> ...
   ```

Another option is the `prefAlignSize` XML attribute of primitive types, this attribute is explained in the chapter Cross-platform options and XML attributes of Primitive types.

7.1.1.3 Contiguous fields declared as unions (nested anonymous unions)

Besides user defined capable classes working as unions, several contiguous fields can be grouped forming an anonymous union, the memory of union fields is shared.

A C example:

```
struct Dimension
{
    int type;
    union
    {
      int x;
      double y;
    };
    const char* desc;
```

```
};
```

The symmetric Java class:

```
package examples.manual;


public class Dimension
{
    protected int type;
    protected int x; // start union
    protected double y;  // end union
    protected String desc;
... /* constructors, get/set methods etc */
}
```

To declare `x` and `y` fields forming an anonymous union, the attribute `union` with values `begin`/`end` must be used in the XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Archive Dimension.jnieasy.enh.xml -->
<jniEasyEnhancer version="1.1">
    <package name="examples.manual">
        <imports />
        <class name="Dimension" type="structure">
            <field name="x" union="begin" />
            <field name="y" union="end" />
        </class>
    </package>
</jniEasyEnhancer>
```

Java "verification":

```
NativeTypeManager typeMgr = JNIEasy.get().getTypeManager();
NativeMultipleFieldContainerDescriptor classDesc =
  (NativeMultipleFieldContainerDescriptor)typeMgr.getClassDescriptor(
```

```
        Dimension.class);
    NativeFieldStructureDescriptor xField =
            (NativeFieldStructureDescriptor)classDesc.getField("x");
    NativeFieldStructureDescriptor yField =
            (NativeFieldStructureDescriptor)classDesc.getField("y");
    System.out.println("Must be 8 (double): " + xField.size());
    System.out.println("Must be 8 (double): " + yField.size());
    System.out.println("Must be true : " + (xField.getOffset() ==
                                            yField.getOffset()));
```

Recommendation: **if a native capable object with union fields is going to be made native owning the native memory, is highly recommended make it native explicitly before setting any field**. The explanation is: when an object is made native owning the native memory the current field values are used to set the native memory in the creation process; the framework has no way to detect in a non native object what fields was and was not set, the latest union field set the native memory state.

Wrong example!!:

```
Dimension dim = new Dimension();
dim.setX(10); // No native memory is set (is non-native)
JNIEasy.get().getNativeManager().makeNative(dim);
// BAD!!!! "y" field with 0 overwrites the "x" field value set
// in the native memory !
System.out.println("Must be true : " + (dim.getX() == 0));
```

Correct example:

```
Dimension dim = new Dimension();
JNIEasy.get().getNativeManager().makeNative(dim);
dim.setX(10); // Sets the native memory too
System.out.println("Must be true : " + (dim.getX() == 10));
```

7.1.2 Declaring Java methods working as proxy of native methods

Methods of Java classes declared as structures, C++ classes or unions can be declared as proxy of C/C++ methods.

C++ methods fit very well with Java methods (instance or static), what about C methods?, there are two options to map Java static methods working as proxies of C methods: use the

code generation utilities (see Java Code Generation), or declare a Java class declared as a (false non-existing) native C++ class and put inside the static Java methods working as proxy of C methods; these "false" native Java classes usually are very useful to group the C methods by categories.

A C++ class example (valid with MSVC and gcc compilers[7]):

```
// MyCPPClassOnDLL.h

class DLLEXPORT MyCPPClassOnDLL

{

protected:

   double m_value;

public:

  MyCPPClassOnDLL(int a,int b);


    static MyCPPClassOnDLL* __stdcall create(int a,int b);


    static void __stdcall destroy(MyCPPClassOnDLL* obj);


    static __int64 __stdcall addStatic(int a,int b);


    double __stdcall getValue();


    virtual double __stdcall sub(int a,int b);


  void __cdecl varargsEx(char* buffer,...);


};
```

[7] All C++ examples of this manual and included into the JNIEasy distribution compile with MSVC 6.0 and a modern gcc (MingGW, cygwin, Linux, Mac and Solaris) compiler, nothing prevent to compile with other compilers like Borland C/C++, Intel C/C++ or Sun Studio, perhaps with minor modifications of `JNIEasy.h` (furthermore, native libraries are language independent, you need to know how native data types are corresponded with Java/JNI data types). The static linking with JNIEasy is only supported with MSVC but can be avoided using `JNIEasy.c` or `JNIEasyHelper.cpp` provided files, static or dynamic linking to JNIEasy DLL is not necessary if `findExportedMethodAddress(const char*)` method is not used from native.

```
// MyCPPClassOnDLL.cpp

#include "MyCPPClassOnDLL.h"
#include <stdio.h>

MyCPPClassOnDLL::MyCPPClassOnDLL(int a,int b)
{
    m_value = a + b;
}

MyCPPClassOnDLL* __stdcall MyCPPClassOnDLL::create(int a,int b)
{
    return new MyCPPClassOnDLL(a , b); // may be an inherited class
}

void __stdcall MyCPPClassOnDLL::destroy(MyCPPClassOnDLL* obj)
{
    delete obj;
}

__int64 __stdcall MyCPPClassOnDLL::addStatic(int a,int b)
{
    return (__int64)a + (__int64)b;
}

double __stdcall MyCPPClassOnDLL::getValue()
{
```

```
    return m_value;
}

double __stdcall MyCPPClassOnDLL::sub(int a,int b)
{
    m_value = m_value - (a + b);
    return m_value;
}

void __cdecl MyCPPClassOnDLL::varargsEx(char* buffer,...)
{
    va_list marker;
    va_start( marker, buffer );
    const char* name = va_arg( marker, const char*);
    int age = va_arg( marker, int);
    int brothers = va_arg( marker, int);
    va_end( marker );

    sprintf(buffer,"%s is %d years old and has %d brothers",name,age,
     brothers);
}
```

If used a Linux/Mac/Solaris gcc compiler, `__int64`, `__cdecl` and `__stdcall` keywords valid with Win32 compilers (MSVC, MinGW gcc, cygwin gcc), are defined in JNIEasy.h as:

```
#define __cdecl   __attribute__((cdecl))
#define __stdcall __attribute__((stdcall))
typedef long long __int64;
```

The `DLLEXPORT` (`__declspec(dllexport)` in Win32 compilers) macro, exports all methods and static fields of the class, this modifier can be used in a per method (and static field) basis. In Visual C++ 6 and gcc 32 bit compilers the presence of a virtual method (the `sub` method) adds

an internal and hidden `int` field: the pointer to the virtual table (is added in the top most class only)[8].

JNI types are not used consciously (`jlong` is `__int64` in Windows, `long long` in Linux/Mac/Solaris x86) to show that JNIEasy integrates with legacy non-Java related code.

The Java class to be used as proxy:

```
package examples.manual;


public class MyCPPClassOnDLL
{
    protected int _virtualTable; // the C++ class has a virtual method
    protected double value;


    public MyCPPClassOnDLL() // mandatory (is not native)
    {
    }


    public MyCPPClassOnDLL(int a, int b)
    {
        throw new RuntimeException("Not enhanced");
    }


    public static void destroy(MyCPPClassOnDLL obj)
    {
        throw new RuntimeException("Not enhanced");
    }
```

[8] In a x86 unix gcc compiler the `MyCPPClassOnDLL` C++ class must be compiled with the flag `-malign-double` because by default the alignment of `double` declared as a field is 4 bytes, this value is the recommended default in old i386 machines, but this alignment is an anachronism and is different to the `double` type size, 8 bytes (primitive sizes and alignments usually share the same value), and incompatible with modern Win32 compilers (MSVC and Win32 gcc ports always align `double` with 8 bytes).

```
    public static native long addStatic(int a,int b);


    public double getValue()   // not a proxy
    {
        return value;
    }


    public double sub(int a,int b)
    {
        throw new RuntimeException("Not enhanced");
    }


    public static void varargsEx(byte[] buffer,Object[] args)
    {
        throw new RuntimeException("Not enhanced");
    }
}
```

JNIEasy supports two forms to declaring a Java native method:

```
    public static void destroy(MyCPPClassOnDLL obj)
    {
        throw new RuntimeException("Not enhanced");
    }


    public static native long addStatic(int a,int b);
```

The body of the first method will be replaced by the enhanced one and a new body will be added to the second method removing the `native` modifier. If the specified `RuntimeException` exception is thrown or a standard Java linker exception is thrown (trying to link the native method) are signs telling us the class was not enhanced, the enhancement process replaces the provided default body and removes the `native` flag if present.

The XML enhancer descriptor (MSVC exported names only):

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Archive MyCPPClassOnDLL.jnieasy.enh.xml -->
<jniEasyEnhancer version="1.1">
    <package name="examples.manual">
        <imports/>
        <class name="MyCPPClassOnDLL" type="class"
                libraryPath="MyLibrary" >

            <constructor onLibrary="true"
                    nativeName="?create@MyCPPClassOnDLL@@SGPAV1@HH@Z"
                 params="int, int" >
            </constructor>

            <method name="destroy" onLibrary="true"
                    nativeName="?destroy@MyCPPClassOnDLL@@SGXPAV1@@Z"
                 params="MyCPPClassOnDLL">
            </method>

            <method name="addStatic" onLibrary="true"
                    nativeName="?addStatic@MyCPPClassOnDLL@@SG_JHH@Z">
                <return />
                <params params="int, int" />
            </method>

            <method name="sub" onLibrary="true"
                    nativeName="?sub@MyCPPClassOnDLL@@UAGNHH@Z">
                <return />
                <params>
```

```
                    <param class="int" />
                    <param class="int" />
                </params>
            </method>

            <method name="varargsEx" onLibrary="true"
                    nativeName="?varargsEx@MyCPPClassOnDLL@@SAXPADZZ"
                 callConv="c_call">
                <return />
                <params>
                    <param class="byte[]" />
                    <param class="Object[]" varargs="true"  />
                </params>
              </method>

        </class>
    </package>
</jniEasyEnhancer>
```

The XML descriptor declares the constructor with two parameters, the `addStatic`, `sub` and `varargsEx` methods as proxies of the corresponding exported methods of the C++ class. The class declares the DLL to be used with the `libraryPath` attribute (a hypothetical `MyLibrary.dll` or `libMyLibrary.so` or `libMyLibrary.dylib`), the .dll, .so and .dylib extensions and the UNIX "lib" prefix may be omitted following the same rules as the `System.loadLibrary` Java method.

The `onLibrary="true"` declares a method as proxy (otherwise is a callback, the default) of a native method and the `nativeName` attribute declares the exported name of the DLL method to link[9]. Several tools can be used to obtain the native name of a DLL/shared object's exported method or field, in Windows use the free tool `PEDUMP`[10] (included inside the distribution), on Linux use `objdump` (with `--dynamic-syms` argument), with Mac use `nm`, and on Solaris use `elfdump` (`-s` argument). On Linux, Mac and Solaris `c++filt` command does the reverse task: demangles an exported name showing the C/C++ original header (very useful in C++), in

[9] The used native names in this manual follow the C++ name mangling of MSVC compiler; in the Cross-platform options chapter we will see how to define a cross-platform native name valid with gcc too. The examples contained into the distribution are fully cross-platform.

[10] http://www.wheaty.net/downloads.htm

Windows Microsoft provides a special method `UnDecorateSymbolName`[11] and the command `undname` included in the Microsoft SDK.

To specify what method/constructor is being enhanced, JNIEasy supports three syntaxes:

```
1) <constructor/method params="class,class,..." >


2) <method ...>
      <return />
         <params params="class,class,..." />
      </method>


3) <method ...>
      <return />
      <params>
         <param class="class" />
         <param class="class" />
      </params>
```

Only the third mode can be used to modify the default native memory layout of the parameter. For instance:

```
<param class="Object[]" varargs="true" />
```

The `param` attribute, `<param>` nodes and the `class` attribute are used to pick the "real" Java class method to be used as proxy; in this example parameters with primitive types are used and the native declaration is straightforward but native information can be added to declare the native view of the parameter as a native variable with the same syntax of the `<field>` tag. The `<return/>` tag works in a similar fashion (note the `class` attribute is optional because the return class is known). The default constructor and `getValue()` method are not declared as proxy in this example, but the `value` field is enhanced and when the `getValue()` Java method is invoked the native C++ field is read too (it works as the C++ `getValue()` version).

If not specified the attribute `callConv`, the default call convention of a method is `std_call`, `callConv` valid values are `std_call` or `c_call`. The C call convention is used in the `varargsEx` method because methods with variable arguments only support C calls. Call convention declared in the Java side and native side must be coincident, only standard and C calls are supported by JNIEasy, any other convention like the "fast call", the default with compilers like MSVC or gcc, is not supported.

The parameter `args` of the method `varargsEx` is declared as "variable number of arguments" (varargs), using the `varargs` attribute set to `true`. A varargs attribute mimics in

[11] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/undecoratesymbolname.asp

Java the C … parameter convention. Without this attribute (or set to false), the default convention of the `Object[]` parameter type is to pass a pointer-to-array to the method, with the `varargs` attribute set to `true`, all array elements are passed as parameters to the method with the "intrinsic" native type.

A Java method working as a proxy does not need a body, because this body is overwritten in enhancement time with a new body calling the native method using an internal `CPPConstructor`, `CMethod`, `CPPMethod` or `CFieldMethod` object attached to the native C/C++ method in the DLL. A good practice is to throw a `RuntimeException` or inherited as method body (the enhancer removes this) to remember the class must be enhanced.

The constructor proxy is a special case because the C++ linked version must be a static method returning the pointer of a new allocated C++ object (**a C++ exported constructor can not be used directly**). When calling the Java proxy constructor, the address returned by the C style constructor is internally used to attach the Java instance to the native C++ object, the Java instance created with a proxy constructor is ever native and attached to the native object, and C++ virtual methods can be called because the C++ related object was created with a normal C++ constructor.

The native static method `addStatic()` is inside a C++ class, this is not a limitation, JNIEasy can link normal C methods, in C++ is recommendable use `extern "C"` to avoid the name mangling of exported C methods in C++ files (or to declare the desired name in a custom .DEF file with MSVC or similar approaches in gcc).

An example of a static method call:

```
byte res = MyCPPClassOnDLL.addStatic(1,2);

System.out.println("Must be 3: " + res);
```

Example of a complete lifecycle of a native instance:

```
MyCPPClassOnDLL obj = new MyCPPClassOnDLL(1,2);

   // Calls create() C++ method. New object is already native

System.out.println("Must be 3: " + obj.getValue());

  // Java method getValue() is not native (field "value" is native)


double res2 = obj.sub(1,2);

System.out.println("Must be 0: " + res2);


MyCPPClassOnDLL.destroy(obj);

         // Calls the C++ method destroy(), the memory is freed
```

An example of varargs call:

```
byte[] buffer = new byte[256];
```

```
MyCPPClassOnDLL.varargsEx(buffer,
    new Object[]{"Joe",new Integer(25),new Integer(2)});
String res = NativePrimitiveUtil.toString(buffer);
System.out.println("Must be true: " +
   "Joe is 25 years old and has 2 brothers".equals(res));
```

The default native type of `"Joe"` is `char*`, and the default native type of `Integer` in a varargs context is the `int` data type (in any other context the related native type is `int*`, to use a pointer to `int` parameter in a varargs call, use a `NativeInteger` object). The method `NativePrimitiveUtil.toString(byte[])` converts a byte based string ended with a 0 element in a `String` object (very useful to work with ANSI strings).

7.1.3 Declaring Java methods working as native callbacks

A Java method declared as a native callback is a normal method (with a signature using native enabled types) callable from native. Selected constructors, static and instance methods in a user defined Java class, declared as a structure, C++ class or union, can be declared as native callbacks.

Example:

```
package examples.manual;

import com.innowhere.jnieasy.core.JNIEasy;


public class MyCPPClassOnJava
{
    protected double value;


    public MyCPPClassOnJava() // mandatory
    {
    }


    public MyCPPClassOnJava(int a,int b)
    {
        this.value = a + b;
        JNIEasy.get().getLockedRegistry().lock(this);
```

```
                        // To avoid GC if called from native
    }

    public static void destroy(MyCPPClassOnJava obj)
    {
        JNIEasy.get().getLockedRegistry().unlock(obj);
                        // To enable GC again
        JNIEasy.get().getNativeManager().free(obj);
    }

    public static long addStatic(int a,int b)
    {
        return (long)a + (long)b;
    }

    public double sub(int a,int b)
    {
        this.value = this.value - (a + b);
        return this.value;
    }
}
```

The XML enhancer descriptor:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Archive MyCPPClassOnJava.jnieasy.enh.xml -->
<jniEasyEnhancer version="1.1">
    <package name="examples.manual">
        <imports />
        <class name="MyCPPClassOnJava" type="class">
            <constructor exportMethod="true" params="int,int" >
```

```
            </constructor>
            <method name="destroy" exportMethod="true"
                params="MyCPPClassOnJava">
            </method>
            <method name="addStatic" exportMethod="true"
                params="int,int">
            </method>
            <method name="sub" exportMethod="true" params="int,int">
            </method>
        </class>
    </package>
</jniEasyEnhancer>
```

The constructor's sentence: `JNIEasy.get().getLockedRegistry().lock(this);` inserts the object with a normal reference in the framework's lock registry to avoid the garbage collector to collect the instance. Why? because a Java instance created from native code does not retain the object with a Java reference (the native call returns a native pointer not a Java reference), after object construction, the object goes out the scope and the garbage collector can free the instance (freeing the allocated native memory) any time; the lock action prevents the GC to free the instance. The method working as a "destructor" unlocks the object, if there is no Java reference pointing the instance it is marked to be garbage collected (an asynchronous "delete").

If the Java object is created from native side, a "real" Java object is created and made native allocating native memory. But if the Java object is created from Java with a normal `new`, no native memory is allocated and the callback methods work as normal; like a native capable Java object it can be made native (allocating or attaching) in any time, and in this state, native fields are synchronized with the native memory.

There are two ways to call Java callbacks from native:

7.1.3.1 Using the internal native method instances

A "native descriptor" of a user defined native capable class can be obtained with the method: `NativeTypeManager.getClassDescriptor(Class)`

Example:

```
NativeTypeManager typeMgr = JNIEasy.get().getTypeManager();
NativeMultipleFieldContainerDescriptor classDesc =
  (NativeMultipleFieldContainerDescriptor)typeMgr.getClassDescriptor(
                          MyCPPClassOnJava.class);
```

```
Constructor construc = MyCPPClassOnDLL.class.getDeclaredConstructor(
                                        new Class[]{int.class,int.class});
NativeConstructorDescriptor constrDesc =
      (NativeConstructorDescriptor)classDesc.getBehavior(construc);
NativeConstructor natConstr = constrDesc.getNativeConstructor();
```

The `natConstr` object is the native constructor instance "making native" the `MyCPPClassOnJava` constructor with two `int` parameters. This native object can be submitted to the native side as a pointer to method.

By default the native method instances are "direct" (`natConstr` is a `NativeDirectConstructorCallback` instance in this case), but the optional `useReflection="true"` attribute (by default is false) can be added to use the reflection version (`NativeConstructorReflection` in this case). Example:

```
<constructor exportMethod="true" useReflection="true"
    params="int,int" >
</constructor>
```

7.1.3.2 Dynamic linking Java methods from C/C++ side querying the addresses with the exported signature.

Exists a C method `findExportedMethodAddress` exported by the JNIEasy DLL.

```
void* __stdcall findExportedMethodAddress(const char* signature)
```

Where `__stdcall` is defined in `JNIEasy.h` with a valid compiler-dependent declaration.

This method returns the address of the method registered in the framework with the specified signature. Any method registered with `DynamicLibrary` methods are automatically exported and accessible using `findExportedMethodAddress` using the following syntax:

```
"[DLLName:]MethodSignature"
```

Where the optional *`DLLName`* is the name used to get the `DynamicLibrary` object from `DLLManager.get(String)` the first time; *`MethodSignature`* follows a mixed Java/C++ syntax, this syntax is explained in the javadoc documentation of the method `NativeBehaviorSignature.getSignatureString(String)`.

The method `findExportedMethodAddress(const char*)` is equivalent to the Java method `JNIEasyLibrary.findExportedMethodAddress(String signature)` (in fact this method is called), this method parses the *`DLLName`*, if declared, and calls `DynamicLibrary.findBehaviorBySignature(String)` using the `DynamicLibrary` object associated to the specified DLL with *`DLLName`*. Example:

```
String sig = ...
NativeBehavior cb =
```

```
JNIEasy.get().getJNIEasyLib().findExportedMethodAddress(sig);
```

If no method was registered with the specified signature returns null.

If the DLL name is omitted the JNIEasy's DLL is supposed, this applies to Java methods as native callbacks, the XML `exportMethod="true"` attribute specifies that the internally used native method object to make native the Java method declared as callback, is exported and accessible with `findExportedMethodAddress`, the native method object is registered with `JNIEasyLibrary.exportBehavior(String nativeName,NativeBehavior method)` using the Java method name as the first parameter (if the method is static the absolute class name is added as prefix and with constructors the `"<init>"` name is ever used).

Any DLL exported method can be found calling `findExportedMethodAddress(const char*)` from native if the exported method was previously registered using an appropriated `DynamicLibrary` method, this "dynamic binding" technique is an alternative to the classic `LoadLibrary/GetProcAddress` in Windows and `dlopen/dlsym` in UNIXes, the main difference is Java exported callbacks can be obtained too (they are "simulated" exported native methods by JNIEasy DLL).

The `findExportedMethodAddress` method is declared in the `JNIEasy.h` public framework file and can be statically linked using `JNIEasy.lib` with MSVC. Is exported by the JNIEasy DLL with the method name `"findExportedMethodAddress"` and can be dynamically linked (in Win32 using `LoadLibrary/GetProcAddress` in Windows and `dlopen/dlsym` in Linux/Mac/Solaris). The files `JNIEasy.c` and `JNIEasyHelper.cpp` are provided to help the dynamic linking process (both files and `JNIEasy.h` can be found in the `include` folder). If `findExportedMethodAddress` is not used your native code has no dependency with JNIEasy (useful macros like `DLLEXPORT` are not directly related with JNIEasy and can be easily copied in your code for your convenience).

The Java class exports the specified callback methods when is loaded and initialized, to ensure a class is loaded and initialized this utility method can be used[12]:

```
NativeCapableUtil.initializeClass(MyCPPClassOnJava.class);
```

or calling `NativeTypeManager.getClass(String)` with the class name, the `ClassLoader` currently defined in JNIEasy will be used to load and initialize the class:

```
JNIEasy.get().getTypeManager().getClass("examples.manual.MyCPPClassOnJava");
```

Following with the example, the signatures of the exported `MyCPPClassOnJava` callbacks are:

```
"examples.manual.MyCPPClassOnJava.<init>(int,int)"

"examples.manual.MyCPPClassOnJava.destroy(examples.manual.MyCPPClassOn
Java*)"

"examples.manual.MyCPPClassOnJava.addStatic(int,int)"

"examples.manual.MyCPPClassOnJava.sub(int,int)"
```

To understand how these signatures are constructed see the javadoc of `JNIEasyLibrary.exportBehavior(String nativeName,NativeBehavior method)`

A C++ class ready to work as a proxy of the Java class may be:

[12] The `Class` object existence indicates the class is loaded but it may be not initialized

```
// MyCPPClassOnJava.h


#include <JNIEasy.h>
#include <JNIEasyHelper.h>


class MyCPPClassOnJava
{
protected:
  static void* (__stdcall * _MyCPPClassOnJava)(int,int);
  static void (__stdcall * _destroy)(void*);
  static __int64 (__stdcall * _addStatic)(int,int);
  static double (__stdcall * _sub)(void*,int,int);
public:
  static MyCPPClassOnJava* create(int a,int b)
  {
    if (_MyCPPClassOnJava == 0)
      _MyCPPClassOnJava = (void* (__stdcall *)(int,int))
     JNIEasyHelper::findExportedMethodAddress(
              "examples.manual.MyCPPClassOnJava.<init>(int,int)");
    return (MyCPPClassOnJava*)_MyCPPClassOnJava(a,b);
  }


  static void destroy(MyCPPClassOnJava* obj)
  {
    if (_destroy == 0)
      _destroy = (void (__stdcall *)(void*))
           JNIEasyHelper::findExportedMethodAddress(
                "examples.manual.MyCPPClassOnJava.destroy(
                     examples.manual.MyCPPClassOnJava*");
```

```
    _destroy(obj);
  }

  static __int64 addStatic(int a,int b)
  {
    if (_addStatic == 0)
      _addStatic = (__int64 (__stdcall *)(int,int))
       JNIEasyHelper::findExportedMethodAddress(
            "examples.manual.MyCPPClassOnJava.addStatic(int,int)");
    return _addStatic(a, b);
  }

  double sub(int a,int b)
  {
    if (_sub == 0)
      _sub = (double (__stdcall *)(void*,int,int))
           JNIEasyHelper::findExportedMethodAddress(
                "examples.manual.MyCPPClassOnJava.sub(int,int)");
    return _sub(this, a, b);
  }
};

// MyCPPClassOnJava.cpp
#include "MyCPPClassOnJava.h"

void* (__stdcall * MyCPPClassOnJava::_MyCPPClassOnJava)(int,int) = 0;
void (__stdcall * MyCPPClassOnJava::_destroy)(void*) = 0;
__int64 (__stdcall * MyCPPClassOnJava::_addStatic)(int,int) = 0;
double (__stdcall * MyCPPClassOnJava::_sub)(void*,int,int) = 0;
```

Example of use:

```
MyCPPClassOnJava* proxy = MyCPPClassOnJava::create(2,3);

      // like a Java: new MyCPPClassOnJava(2,3);

double res = proxy->sub(2,3); // Must be 0

MyCPPClassOnJava::destroy(proxy); // Unlock the Java object
```

C++ objects created as proxy of Java objects (using the Java constructor) are not created with the normal C++ `new` sentence, the Java side allocates the native memory, then the C++ `delete` sentence must not be used to destroy a Java allocated object (the C++ `new` was not used), the Java native instance will free the native memory.

Of course a C++ object can be constructed with a normal C++ constructor. Following the example:

```
// MyCPPClassOnJava.h

class MyCPPClassOnJava

{

protected:

  double m_value;  // the same as in Java class

. . .

public:

  MyCPPClassOnJava(int a,int b)  // the same as in Java class

  {

    m_value = a + b;

  }

. . .

};
```

Example of use:

```
MyCPPClassOnJava* proxy = new MyCPPClassOnJava(2,3);

double res = proxy->sub(2,3); // Must be 0

delete proxy;
```

This example do the same as the previous, but it works in a different manner: only the method `sub` is working as proxy and calling the related Java method, when invoking, a temporal `MyCPPClassOnJava` Java instance is created and attached to the address of the C++ object and the Java `sub` method is called, the Java field `value` accessed inside the method is synchronized with the native memory, the read/write change of the Java field is performed with the native

memory field, the returned value is the expected. This operational mode (allocating memory in the C++ side) is not performant because a new Java instance is created and attached to native object address in a per C++ proxy/Java method call basis, the temporal Java instance is discarded in the end of call. The other operational mode, calling a Java constructor (allocating memory in the Java side) is more performant because the Java instance is unique it is internally registered as a native memory owner (reminder: this registry does not hold the object with a strong reference) and locked in the constructor (with a strong reference).

A note about Mac OS X 10.4 (Tiger) gcc 4.0 and `findExportedMethodAddress`: gcc on Mac makes aggressive optimizations using the flag -Ox (x != 0), for instance the "omit-frame-pointer" optimization. A call to `findExportedMethodAddress` enters inside the Java VM, if you are experimenting strange problems try to add `-fno-omit-frame-pointer` to the gcc compiler options or remove the –Ox flag.

7.1.3.3 Java fields working as native field (method) callbacks

The Java fields, static or instance, of native capable classes are a special case of callbacks. A non-native Java field can be wrapped, made native and exported to the native world with a related field-method callback created by the framework behind the scenes.

To do this, is necessary to add a `<fieldMethod>` declaration in the XML enhancer file, declaring the field name and the native type declaration of the field type.

In the previous XML file `MyCPPClassOnJava.jnieasy.enh.xml`:

```
...
  <fieldMethod name="value" exportMethod="true" />
</class>
...
```

The new tag declares the field `value` as a field-method (in this case the field is already declared native but this is not mandatory), using the default native related type of `int` and the standard call convention (`std_call`), the method is exported and can be obtained using the method `findExportedMethodAddress(const char*)`. By default the native field-method instance is "direct" (`NativeDirectInstanceFieldCallback` in this case), but the optional `useReflection="true"` attribute can be added to use the reflection version (`NativeInstanceFieldMethodReflection` in this case):

```
<fieldMethod name="value" exportMethod="true" useReflection="true" />
```

The native field-method instance to be used can be obtained as any other native method with the native class descriptor, but using the `java.lang.reflect.Field` object of the field.

```
NativeTypeManager typeMgr = JNIEasy.get().getTypeManager();
CPPClassDescriptor classDesc =
   (CPPClassDescriptor)typeMgr.getClassDescriptor(MyCPPClassOnJava.class);
Field field = MyCPPClassOnJava.class.getDeclaredField("value");
NativeBehaviorDescriptor methodDesc = classDesc.getBehavior(field);
```

If not using reflection:

```
NativeDirectInstanceFieldCallback callback =
        (NativeDirectInstanceFieldCallback)methodDesc.getNativeBehavior();
```

If using reflection:

```
NativeInstanceFieldMethodReflection callback =
 (NativeInstanceFieldMethodReflection)methodDesc.getNativeBehavior();
```

The signature to use with `findExportedMethodAddress` is:

```
"examples.manual.MyCPPClassOnJava.value(int,double)"
```

To use this exported method from native:

```
// MyCPPClassOnJava.h
class MyCPPClassOnJava
{
. . .
  static double (__stdcall * _value)(void*,int,double);
. . .
  double MyCPPClassOnJava::valueField(int opcode,double value)
  {
      if (_value == 0)
     _value = (double (__stdcall *)(void*,int,double))
                JNIEasy::findExportedMethodAddress(
          "examples.manual.MyCPPClassOnJava.value(int,double)");

   return _value(this,opcode,value);
  }

};

// MyCPPClassOnJava.cpp
double (__stdcall * MyCPPClassOnJava::_value)(void*,int,double) = 0;
```

An example of call in C++:

```
MyCPPClassOnJava* proxy = ...

proxy->valueField(NativeFieldMethod::SET,10);

double res = proxy->valueField(NativeFieldMethod::GET,0);

// Must be 10
```

The `NativeFieldMethod` class (holding the `GET`, `SET` and `GET_SET` constants) is defined in `JNIEasy.h`, constants 0 (`GET`), 1 (`SET`) and 2 (`GET_SET`) can be used too.

Important: a native field-method callback access the related Java field directly (using a direct access or reflection), if the (instance) field is native the native memory is not read or modified. A possible new value set with the native field-method can be lost and updated with the native memory value when reading the Java field normally, and the Java field value read with the native field-method is not ensured to be synchronized with the native memory. In our example the Java field is already native and can be accessed directly from Java (using get and set methods). Is not recommended to use a Java field declared as native and accessed using a field-method (the Java field can be read/updated without the native memory synchronization)

The export of a Java field as a method is especially useful with static fields (a static field can not be declared as native directly) and non-native instance fields.

7.1.4 Mixing models: a Java class working as proxy and callback

The proxy/callback modeling is based on methods not classes; there is no limitation to mix in the same Java class methods working as proxy and methods working as callbacks.

7.1.5 Inheritance

Java native capable classes, declared as C++ classes and structures, support inheritance; they can inherit from another Java native capable class mapping the C++ tree. C++ unions do not support inheritance, this is applied to Java unions too.

In enhancement time JNIEasy automatically detects if a user defined Java native capable class inherits from another native capable class, there is no declaration in the XML descriptor. The framework ever enhances first the base native capable class locating and loading the XML descriptor of the base class, the XML file must be alongside the .class file. Is not allowed to insert in the middle of a native capable inheritance tree a non-native capable class (use a native capable class without native fields if you need something similar), but is valid to inherit from a non-native capable class as the top most base class (in fact all classes inherit from `java.lang.Object`).

7.1.5.1 Inheritance in Java classes working as proxies

There is a limitation regarding C++ virtual methods (declared with the `virtual` keyword): an exported virtual method (like `sub`) is called from Java ignoring the virtual rule because its method address is used directly (is not obtained using the virtual method table). This is a minor limitation because **the virtual behavior works in the Java side**, the Java method called (supposed not `final`) is the most derived, and this method calls the related derived method in the C++ side.

7.1.5.2 Inheritance in Java classes working as callbacks

There is a limitation: C++ objects created as proxy of a Java object (using the Java constructor) are not created with the normal C++ `new` sentence and no C++ constructor is called, a C++ constructor initializes the virtual table of methods (if any), wherefore C++ virtual methods (declared with "`virtual`" keyword) can not be called. This is a minor limitation because **the virtual behavior works in the Java side** and the most derived method is called. Base and derived methods must be declared in the C++ proxy without the `virtual` keyword, regardless the C++ method called the most derived Java method is called. Of course C++ virtual methods works ok if the object is created with a `new` sentence invoking a normal C ++ constructor.

7.1.6 Inner classes

Inner classes are allowed as native capable classes, but only static inner classes. The XML enhancer descriptor file must follow the same name rules as the .class file. For instance, if the `MyStructure` class declares a static inner class named `MyInnerStructure`, the XML file name must be:

```
MyStructure$MyInnerStructure.jnieas.enh.xml
```

And the XML content may be:

```
<?xml version="1.0" encoding="UTF-8"?>
<jniEasyEnhancer version="1.1">
    <package name="examples.manual">
        <imports/>
        <class name="MyStructure$MyInnerStructure" type="structure" />
    </package>
</jniEasyEnhancer>
```

7.2 USER DEFINED NATIVE CAPABLE DIRECT CALLBACKS

The developer can export to the native side, single Java methods with native capable signatures outside Java classes as C++ classes, structures and union. These methods, of course, must be declared inside Java classes (a Java imperative). A user defined Java class containing a native capable method may be declared as a native capable direct callback; a native instance has an address and represents a native method with a native signature callable from the native side (a pointer-to-method); in fact if the native instance is freed, explicitly or by the garbage collector, the native method can not be called from native because the method address is not valid (and can be reused by another native element). A native method instance must be seen as a native method compiled "on the fly". Two native instances of the same class (same method) have different addresses, they must be considered like two different methods.

7.2.1 Static callbacks

A simple example with a static method:

```
package examples.manual;


public class AddTwoNumbersCallback
{
    public static long add(int a,int b)
    {
        return (long)a + (long)b;
    }
}
```

The XML enhancer file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- AddTwoNumbersCallback.jnieasy.enh.xml -->
<jniEasyEnhancer version="1.1">
    <package name="examples.manual">
        <imports />
        <class name="AddTwoNumbersCallback" type="callback" >
            <method name="add" params="int,int" >
            </method>
        </class>
    </package>
</jniEasyEnhancer>
```

The signature in the C/C++ side may be:

```
__int64 (__stdcall *) (int,int)
```

A native instance can be used in the scenarios where a method pointer with this signature is required, in fact, a native instance itself is a native pointer-to-method in Java much like a `CMethod`/`CPPConstructor`/`CPPMethod`/`CFieldMethod` mapping a DLL method, the Java method can be called from Java with the native marshalling.

An example of native call from Java:

```
NativeStaticMethodCallback cb =
                (NativeStaticMethodCallback)new AddTwoNumbersCallback();
```

```
JNIEasy.get().getNativeManager().makeNative(cb);

long res = cb.callLong(new Object[]{new Integer(2),new Integer(3)});

System.out.println("Must be 5: " + res);
```

The method `AddTwoNumbersCallback.add(in,int)` is called using a native bridge. This feature is useful to test any Java native callback from Java.

In the native programming world is very common to use pointer-to-method parameters used as callbacks to be called into the method invoked, the C method signature is working like an "interface", and concrete C methods can be used as parameters. In the previous example the wrapped method was static, the user defined callback class name works as the native C signature of the method, but the type and concrete method are "welded", the user defined callback can not be used as parameter type of "implementation changeable" callback parameters of native methods declared in Java, because the callback implementation of any native callback instance is ever the same. To avoid this problem the wrapped method may be non-static, but the signature may remain as static too, a non-static method can be overwritten in an inherited class, the only limitation is the non-static method can not be abstract (neither the class).

Developing a version "to override":

```
package examples.manual;


public class AddTwoNumbersCallbackToOverride
{
    public AddTwoNumbersCallbackToOverride()
    {
    }

    public long add(int a,int b)
    {
        throw new RuntimeException("Override this method");
    }
}
```

The XML enhancer file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- AddTwoNumbersCallbackToOverride.jnieasy.enh.xml -->
<jniEasyEnhancer version="1.1">
```

```
        <package name="examples.manual">
            <imports />
            <class name="AddTwoNumbersCallbackToOverride" type="callback" >
                <method name="add" params="int,int" >
                </method>
            </class>
        </package>
    </jniEasyEnhancer>
```

Note the `AddTwoNumbersCallbackToOverride` XML file is identical to the `AddTwoNumbersCallback` version, in both cases the `add` method is a static native method, but the native instance is used to call the `add` instance method.

In our example is not useful to create instances of `AddTwoNumbersCallbackToOverride`, because the `add` method does nothing (throws an exception), but the class name can be used now as a "placeholder" of concrete implementations, native Java methods now can use this class as parameter type and expect objects with derived classes. Of course another approach is to use fields to distinguish the method behavior in a per instance basis, but this approach is not so elegant.

Derived classes do not need to be enhanced, no XML enhancer descriptor is needed, and instances of these classes are native capable because the base class is already native capable.

Following the example:

```
package examples.manual;


public class AddTwoNumbersCallbackConcrete extends
                                        AddTwoNumbersCallbackToOverride
{
    public AddTwoNumbersCallbackConcrete()
    {
    }

    public long add(int a,int b)
    {
        return (long)a + (long)b;
    }
```

```
}
```

Instances of this class can be made native and passed as parameters to native methods where a `AddTwoNumbersCallbackToOverride` object is expected.

A real world example, the `EnumWindows` method of the Win32 API (Microsoft documentation):

"The EnumWindows function enumerates all top-level windows on the screen by passing the handle to each window, in turn, to an application-defined callback function. EnumWindows continues until the last top-level window is enumerated or the callback function returns FALSE"[13].

The C signature is the following:

```
BOOL EnumWindows(
        WNDENUMPROC lpEnumFunc,   // pointer to callback function
        LPARAM lParam             // application-defined value
       );
```

where `WNDENUMPROC` is defined as:

```
typedef BOOL (CALLBACK* WNDENUMPROC)(HWND, LPARAM);
```

and `CALLBACK` is defined as:

```
#define CALLBACK    __stdcall
```

The Java user defined callback "template":

```
package examples.manual;


public class EnumWindowsProc
{
    public EnumWindowsProc()
    {
    }


    public int onCall(int hwnd,int lParam)
    {
        throw new RuntimeException("Override this method");
```

[13]http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winui/winui/windowsuserinterface/windowing/windows/windowreference/windowfunctions/enumwindows.asp

```
        }
    }
```

The XML enhancer file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- EnumWindowsProc.jnieasy.enh.xml -->
<jniEasyEnhancer version="1.1">
    <package name="examples.manual">
        <imports />
        <class name="EnumWindowsProc" type="callback" >
            <method name="onCall" callConv="std_call"
                params="int,int" >
            </method>
        </class>
    </package>
</jniEasyEnhancer>
```

The concrete callback (no need of a XML descriptor):

```
package examples.manual;


public class EnumWindowsProcConcrete extends EnumWindowsProc
{
    public int windowCount;


    public EnumWindowsProcConcrete()
    {
    }


    public int onCall(int hwnd, int lParam)
    {
        windowCount++;
```

```
            return 1; // TRUE
        }

        public int getWindowCount()
        {
            return windowCount;
        }
    }
```

and finally an example of use:

```
DynamicLibrary dll = JNIEasy.get().getDLLManager().get("User32");
CMethod method = dll.addCMethod("EnumWindows",int.class,
  new Object[]{EnumWindowsProc.class,int.class},CallConv.STD_CALL);
EnumWindowsProc lpEnumFunc = new EnumWindowsProcConcrete();
int res = method.callInt(new Object[]{lpEnumFunc,new java.lang.Integer(5)});
System.out.println("Opened windows :" + lpEnumFunc.getWindowCount());
```

7.2.2 Static callbacks with the method outside the class

In previous examples, the exported method was inside the native capable callback class, this is not mandatory, the method can be outside the callback class and in the case of static methods the container class does not need to be a native capable class.

Example:

```
package examples.manual;

public class AddTwoNumbersCallbackOutside
{
    public AddTwoNumbersCallbackOutside()
    {
    }
}
```

The XML enhancer descriptor:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- AddTwoNumbersCallbackOutside.jnieasy.enh.xml -->
<jniEasyEnhancer version="1.1">
    <package name="examples.manual">
        <imports />
        <class name="AddTwoNumbersCallbackOutside" type="callback" >
            <method name="addStatic" thisClass="MyCPPClassOnJava"
                params="int,int">
            </method>
        </class>
    </package>
</jniEasyEnhancer>
```

The attribute `thisClass="MyCPPClassOnJava"` specifies the class where the method (`addStatic`) must be searched.

7.2.3 Instance callbacks

A user defined callback can make native an instance Java method seen as instance method in the native side too. This method must be declared in a user defined native capable class declared as C++ class, structure or union ("field based" user defined native capable classes). This (native capable) method does not need to be declared as native in the container class's XML enhancer descriptor because the native method view is managed by the callback class, anyway a native instance of the container's class is needed to call this method from native (because is a instance/non-static method in Java and in native side). The method is invoked from native using an instance of the container class as the first parameter (using a C form layout of the instance method with a container class pointer as the first parameter).

Example: mapping the `sub` method of `MyCPPClassOnJava`

The Java callback:

```
package examples.manual;


public class MyCPPClassOnJavaSubCallback
{
    public MyCPPClassOnJavaSubCallback()
    {
    }
```

```
}
```

The XML descriptor:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- MyCPPClassOnJavaSubCallback.jnieasy.enh.xml -->
<jniEasyEnhancer version="1.1">
    <package name="examples.manual">
        <imports />
        <class name="MyCPPClassOnJavaSubCallback" type="callback" >
            <method name="sub" thisClass="MyCPPClassOnJava"
                    params="int,int" >
            </method>
        </class>
    </package>
</jniEasyEnhancer>
```

The `thisClass` attribute specifies the container class of `sub`.

One step forward: declaring a constructor and `destroy` callbacks we have an example of a complete lifecycle of a `MyCPPClassOnJava` native instance without using `MyCPPClassOnJava` explicitly.

Mapping the two integer constructor with a Java callback:

```
package examples.manual;


public class MyCPPClassOnJavaConstructorCallback
{
    public MyCPPClassOnJavaConstructorCallback()
    {
    }
}
```

The XML enhancer descriptor of the constructor:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!-- MyCPPClassOnJavaConstructorCallback.jnieasy.enh.xml -->
<jniEasyEnhancer version="1.1">
    <package name="examples.manual">
        <imports />
        <class name="MyCPPClassOnJavaConstructorCallback"
                type="callback" >
            <constructor thisClass="MyCPPClassOnJava"
                params="int,int">
            </constructor>
        </class>
    </package>
</jniEasyEnhancer>
```

Mapping the `destroy` method with a Java callback:

```
package examples.manual;


public class MyCPPClassOnJavaDestroyCallback
{
    public MyCPPClassOnJavaDestroyCallback()
    {
    }
}
```

The XML enhancer descriptor of the `destroy` callback:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- MyCPPClassOnJavaDestroyCallback.jnieasy.enh.xml -->
<jniEasyEnhancer version="1.1">
  <package name="examples.manual">
     <imports />
     <class name="MyCPPClassOnJavaDestroyCallback" type="callback" >
```

```
            <method name="destroy" thisClass="MyCPPClassOnJava"
                    params="MyCPPClassOnJava">
            </method>
        </class>
    </package>
</jniEasyEnhancer>
```

Putting it all together:

```
NativeConstructorCallback constCb =
  (NativeConstructorCallback)new MyCPPClassOnJavaConstructorCallback();
JNIEasy.get().getNativeManager().makeNative(constCb);
Object[] args = new Object[] { new Integer(2), new Integer(3) };
Object obj = constCb.call(args); // Calls constructor and
                        // returns a native MyCPPClassOnJava instance
NativeInstanceMethodCallback cb =
      (NativeInstanceMethodCallback)new MyCPPClassOnJavaSubCallback();
JNIEasy.get().getNativeManager().makeNative(cb);
args = new Object[] { new Integer(2), new Integer(3) };
double res = cb.callDouble(obj,args); // Calls sub(int,int)


NativeStaticMethodCallback destCb =
     (NativeStaticMethodCallback)new MyCPPClassOnJavaDestroyCallback();
JNIEasy.get().getNativeManager().makeNative(destCb);
destCb.callVoid(new Object[]{obj}); // Calls destroy(MyCPPClassOnJava)
```

7.2.4 Exporting the callbacks to the native side

The `exportMethod="true"` can be used in the declaration of a user defined native capable callback, if set, a native instance callback is created, made native and exported when the callback class is loaded.

Example: updating the `AddTwoNumbersCallback` XML file:

```
    ...
```

```
<class name="AddTwoNumbersCallback" type="callback" >

   <method name="add" exportMethod="true" >

...
```

In the native side the callback pointer can be obtained using `findExportedMethodAddress` and the signature name:

```
"examples.manual.AddTwoNumbersCallback.add(int,int)"
```

Example:

```
JNIEasyLibrary dll = JNIEasy.get().getJNIEasyLib();

String sig = "examples.manual.AddTwoNumbersCallback.add(int,int)";

NativeStaticMethod method =

                (NativeStaticMethod)dll.findExportedMethodAddress(sig);

long res = method.callLong(new Object[]{new Integer(1),new Integer(2)});

// add(1,2) call

System.out.println("Must be 3: " + res);
```

7.3 USER DEFINED POINTERS

The `NativePointer` interface is implemented by native capable classes wrapping a native capable or "can be native" object reference working as a native pointer. A `NativePointer` native instance holds a native pointer, wherefore a `NativePointer` native reference is like a C/C++ "pointer to pointer" (ex. `int**`).

The framework offers a default implementation, therefore a `NativePointer` field can be declared as a native field in a structure etc; the internal pointer (which type is declared in the XML descriptor of the field) can be got /set with the methods `getValue()/setValue(Object)`.

The developer can code user defined native capable class pointers holding a field with a fixed pointer type. These user defined classes can be used instead of the default implementation.

Example: a pointer to an UNICODE `String` (`wchar_t**`)

```
package examples.manual;


public class PointerToString

{

    protected String ptrToStr;
```

```
    public PointerToString() // mandatory
    {
    }

    public PointerToString(String pointer)
    {
        this.ptrToStr = pointer;
    }

    public String getString()
    {
        return ptrToStr;
    }

    public void setString(String pointer)
    {
        this.ptrToStr = pointer;
    }
}
```

The XML enhancer file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Archive PointerToString.jnieasy.enh.xml -->
<jniEasyEnhancer version="1.1">
    <package name="examples.manual">
        <imports/>
        <class name="PointerToString" type="pointer">
            <field name="ptrToStr" encoding="unicode" />
        </class>
    </package>
```

```
</jniEasyEnhancer>
```

A `PointerToString` reference is seen as a `char**` C type.

7.4 USER DEFINED NATIVE CAPABLE ARRAYS

Java arrays need native capable classes to manage its native state. All combinations of Java arrays of enabled types are managed by the framework, including multidimensional arrays. Most of them have a specific interface/default native capable class with no need of casting.

There are two families of arrays with a default implementation managing multiple types (casting needed):

1. Multidimensional arrays (ex. `int[][]`)
2. Arrays of native capable classes (ex. `MyStructure[]` or `NativeString[]`)

The developer can code user defined native capable class arrays holding a concrete Java array field: a native capable or multidimensional array. These user defined classes can be used instead of the default implementation.

7.4.1 Multidimensional arrays

Example: a two-dimensional `int` array

```
package examples.manual;


import com.innowhere.jnieasy.core.data.NativeObjectArray;


public class IntArray2Dim
{
    private int[][] value;


    public IntArray2Dim() // mandatory
    {
    }


    public IntArray2Dim(int[][] value)
    {
        this.value = value;
```

```
}

public int[][] getIntArray()
{
    return value;
}

public void setIntArray(int[][] value)
{
    this.value = value;
}

public int getInt(int[] dims)
{
    return this.value[dims[0]][dims[1]];
}

public void setInt(int[] dims,int newValue)
{
    int[][] aux = this.value;
    aux[dims[0]][dims[1]] = newValue;
    this.value = aux;
}

public int getIntQuick(int[] dims)
{
    NativeObjectArray thisArr = (NativeObjectArray)this;
    return ((Integer)thisArr.getElement(dims)).intValue();
}
```

```
    public void setIntQuick(int[] dims,int newValue)
    {
        NativeObjectArray thisArr = (NativeObjectArray)this;
        thisArr.setElement(dims,new Integer(newValue));
    }
}
```

The XML enhancer file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Archive IntArray2Dim.jnieasy.enh.xml -->
<jniEasyEnhancer version="1.1">
    <package name="examples.manual">
        <imports/>
        <class name="IntArray2Dim" type="array">
            <field name="value" />
        </class>
    </package>
</jniEasyEnhancer>
```

The field `value` is internally and automatically declared "by value" (the native memory layout is the same as a structure with a single array field by value), and can not be declared "by pointer".

The native capable array container can have any values of dimensions; a concrete instance and field value imposes the max values of dimensions (and memory size). To declare fixed values (lengths) of dimensions:

```
        . . .
        <field name="value" length="2">
            <component length="3" />
        </field>
        . . .
```

The previous example declares the custom class as a fixed size array, all instances have the same maximum lengths (2 and 3).

7.4.1.1 Review of the method `getInt(int[])`

The methods `getInt(int[])` and `setInt(int[],int)` are examples of methods to get and set an element of the array ensuring synchronization with the native memory, the enhancer can not control an element access of an array.

```
return this.value[dims[0]][dims[1]];
```

Reads the native memory and copies the array in `this.value`, then returns the concrete element read from Java memory.

7.4.1.2 Review of the method `setInt(int[],int)`

```
int[][] aux = this.value;
```

The `aux = this.value;` reads the native memory and copies the array in `this.value`, the `array` reference points to the Java array.

```
aux[dims[0]][dims[1]] = newValue;
```

Sets the new element in the array on Java memory.

```
this.value = aux;
```

Updates the `this.value` field and the native memory of the whole array.

7.4.1.3 Review of the "quick" methods

The methods `getIntQuick(int[])` and `setIntQuick(int[],int)` are the speeder versions of the previous methods, only the strictly necessary native memory is read/written (the element read/updated). The multidimensional array is a `NativeObjectArray` after the enhancement process. Of course `NativeObjectArray` methods can be called outside the class.

7.4.2 Arrays of native capable classes

Example: an array of structures (elements declared "by pointer").

```
package examples.manual;


import com.innowhere.jnieasy.core.data.NativeObjectArray;


public class MyStructureArray
{
    private MyStructure[] value;


    public MyStructureArray()
```

```
    {
    }

    public MyStructureArray(MyStructure[] value)
    {
        this.value = value;
    }

    public MyStructure[] getStructArray()
    {
        return value;
    }

    public void setStructArray(MyStructure[] value)
    {
        this.value = value;
    }

    public MyStructure getStruct(int index)
    {
        return value[index];
    }

    public void setStruct(int index,MyStructure value)
    {
        MyStructure[] aux = this.value;
        aux[index] = value;
        this.value = aux;
    }
```

```
    public MyStructure getStructQuick(int index)
    {
        NativeObjectArray thisArr = (NativeObjectArray)this;
        return (MyStructure)thisArr.getElement(index);
    }

    public void setStructQuick(int index,MyStructure value)
    {
        NativeObjectArray thisArr = (NativeObjectArray)this;
        thisArr.setElement(index,value);
    }
}
```

The XML enhancer file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Archive MyStructureArray.jnieasy.enh.xml -->
<jniEasyEnhancer version="1.1">
    <package name="examples.manual">
        <imports/>
        <class name="MyStructureArray" type="array">
            <field name="value" />
        </class>
    </package>
</jniEasyEnhancer>
```

Again the field `value` is internally declared "by value", the native capable array container can have any values of dimensions (a fixed array length can be specified too with the `length` attribute); array elements (structures) are declared "by pointer" (the default native variable type), then `MyStructureArray` is an array of `MyStructure` pointers (`MyStructure*[]`).

7.5 MAPPING NATIVE LEGACY CLASSES

We have seen how a C++ class can be accessed from Java using JNIEasy, basically exporting the required methods and adding standard or C call conventions. What about a legacy C++ class? If we cannot touch the source code we can ever add special C methods wrapping and exporting legacy methods. In C++ an instance method is called passing as first parameter a pointer to the object instance, an instance method is basically a C method with a hidden parameter (a pointer to the object), a virtual table is used to resolve the address of the method. We can simulate with C methods false C++ methods, of course excluding virtual addressing.

Example:

Legacy C++ class `MyLegacyClassOnDLL` (header):

```
#ifndef MyLegacyClassOnDLL_h

#define MyLegacyClassOnDLL_h


#include "JNIEasy.h"


// MyLegacyClassOnDLL.h

class MyLegacyClassOnDLL

{

protected:

    double m_value;


public:

    MyLegacyClassOnDLL(int a,int b);

    virtual ~MyLegacyClassOnDLL();


    static __int64 addStatic(int a,int b);


    double getValue();


    virtual double sub(int a,int b);

};


extern "C"
```

```
{
DLLEXPORT MyLegacyClassOnDLL* __stdcall MyLegacyClassOnDLL_create(
                                                        int a,int b);
DLLEXPORT void __stdcall MyLegacyClassOnDLL_destroy(
                                        MyLegacyClassOnDLL* obj);
DLLEXPORT __int64 __stdcall MyLegacyClassOnDLL_addStatic(int a,int b);
DLLEXPORT double __stdcall MyLegacyClassOnDLL_getValue(
                                        MyLegacyClassOnDLL* obj);
DLLEXPORT double __stdcall MyLegacyClassOnDLL_sub(
                               MyLegacyClassOnDLL* obj,int a,int b);
}

#endif
```

Implementation:

```
#include "MyLegacyClassOnDLL.h"
#include <stdio.h>
#include <stdarg.h>

MyLegacyClassOnDLL::MyLegacyClassOnDLL(int a,int b)
{
    m_value = a + b;
}

MyLegacyClassOnDLL::~MyLegacyClassOnDLL()
{
}

__int64 MyLegacyClassOnDLL::addStatic(int a,int b)
{
```

```
    return (__int64)a + (__int64)b;
}

double MyLegacyClassOnDLL::getValue()
{
    return m_value;
}

double MyLegacyClassOnDLL::sub(int a,int b)
{
    m_value = m_value - (a + b);
    return m_value;
}

// Wrapper Methods

MyLegacyClassOnDLL* __stdcall MyLegacyClassOnDLL_create(int a,int b)
{
    return new MyLegacyClassOnDLL(a , b); // may be an inherited class
}

void __stdcall MyLegacyClassOnDLL_destroy(MyLegacyClassOnDLL* obj)
{
    delete obj;
}

__int64 __stdcall MyLegacyClassOnDLL_addStatic(int a,int b)
{
    return MyLegacyClassOnDLL::addStatic(a,b);
```

```
}

double __stdcall MyLegacyClassOnDLL_getValue(MyLegacyClassOnDLL* obj)
{
    return obj->getValue();
}

double __stdcall MyLegacyClassOnDLL_sub(MyLegacyClassOnDLL* obj,
                                        int a,int b)
{
    return obj->sub(a,b);
}
```

C methods are simple wrappers of the C++ methods exporting them to Java. Original class fields can be directly mapped in Java.

The Java class:

```
public class MyLegacyClassOnDLL
{
    protected int virtualTable; // the C++ class has a virtual method
    protected double value;

    public MyLegacyClassOnDLL() // mandatory (is not native)
    {
    }

    public MyLegacyClassOnDLL(int a, int b)
    {
        throw new RuntimeException("Not enhanced");
    }
```

```
    public static native void destroy(MyLegacyClassOnDLL obj);


    public static native long addStatic(int a,int b);


    public double getValue()
    {
        return value;
    }


    public native double sub(int a,int b);
}
```

The XML file, `MyLegacyClassOnDLL.jnieasy.enh.xml`, for the enhancer (is added to the enhancer XML descriptor with the list to enhance too):

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Archive MyLegacyClassOnDLL.jnieasy.enh.xml -->
<jniEasyEnhancer version="1.1" ...>


    <package name="examples.manual">
        <imports/>
        <class name="MyLegacyClassOnDLL" type="class"
                libraryPath="MyLibrary" >


            <constructor onLibrary="true"
nativeName="MSC:_MyLegacyClassOnDLL_create@8;gcc:MyLegacyClassOnDLL_cr
eate" params="int, int">
            </constructor>


            <method name="destroy" onLibrary="true"
```

```
nativeName="MSC:_MyLegacyClassOnDLL_destroy@4;gcc:MyLegacyClassOnDLL_d
estroy" params="MyLegacyClassOnDLL">

            </method>


            <method name="addStatic" onLibrary="true"

nativeName="MSC:_MyLegacyClassOnDLL_addStatic@8;gcc:MyLegacyClassOnDLL
_addStatic" params="int,int">

            </method>


            <method name="sub" onLibrary="true"

nativeName="MSC:_MyLegacyClassOnDLL_sub@12;gcc:MyLegacyClassOnDLL_sub"

                    params="int,int">

            </method>


        </class>

    </package>

</jniEasyEnhancer>
```

And finally a use example:

```
byte res = MyLegacyClassOnDLL.addStatic(1,2);

System.out.println("Must be 3: " + res);


MyLegacyClassOnDLL obj = new MyLegacyClassOnDLL(1,2);

   // Calls create() C++ method. New object is already native

System.out.println("Must be 3: " + obj.getValue());

  // Java method getValue() is not native (field "value" is native)


double res2 = obj.sub(1,2);

System.out.println("Must be 0: " + res2);
```

```
MyLegacyClassOnDLL.destroy(obj);

        // Calls the C++ method destroy(), the memory is freed
```

8. ENHANCER

User defined classes must be enhanced to be converted in native capable classes. The enhancer adds new code and modifies the byte code of the compiled class providing the transparent behavior of the native memory-Java memory synchronization.

Enhancement can be done on filesystem or on runtime (on class loading).

8.1 ENHANCEMENT DONE ON FILESYSTEM

If the enhancement is done on filesystem, this task does not use the native memory and/or load user dynamic link libraries (except the framework's DLL).

An enhanced .class in the filesystem does not need the XML enhancer descriptor file in runtime and can be distributed without it.

The enhancement process of a user defined class may need to create new (internal) native capable classes, theses classes are saved alongside the user defined .class; on runtime these classes are loaded automatically when necessary.

The enhancement task is usually performed using the `NativeEnhancerCmd` class in the command line or Ant task, or using the method:

```
NativeEnhancer.enhance(URL filePath,String outputDir)
```

In both cases, the enhancement is made in "batch" mode using a "root" XML file listing the XML descriptor files of classes to be enhanced. The name of this "root" XML file is not fixed.

The following example file (`enhancer.xml`) list all XML enhancer descriptor files of all user defined native capable classes used in this manual, all XML files are in the same directory in this example:

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- enhancer.xml -->

<jniEasyEnhancer version="1.1">

    <include file="AddTwoNumbersCallback.jnieasy.enh.xml" />

    <include file="AddTwoNumbersCallbackOutside.jnieasy.enh.xml" />

    <include file="AddTwoNumbersCallbackToOverride.jnieasy.enh.xml" />

    <include file="Dimension.jnieasy.enh.xml" />

    <include file="EnumWindowsProc.jnieasy.enh.xml" />

    <include file="IntArray2Dim.jnieasy.enh.xml" />

    <include file="MyCPPClassOnDLL.jnieasy.enh.xml" />

    <include file="MyCPPClassOnJava.jnieasy.enh.xml"/>
```

```
    <include
file="MyCPPClassOnJavaConstructorCallback.jnieasy.enh.xml"/>

    <include file="MyCPPClassOnJavaDestroyCallback.jnieasy.enh.xml"/>

    <include file="MyCPPClassOnJavaSubCallback.jnieasy.enh.xml"/>

    <include file="MyStructure.jnieasy.enh.xml"/>

    <include file="MyStructureArray.jnieasy.enh.xml"/>

    <include file="PointerToString.jnieasy.enh.xml"/>


    <include file="inc/enhancer2.xml" />

</jniEasyEnhancer>
```

The `file` attribute of the `include` tag expects the relative file path, or the URL of a XML enhancer descriptor class file or another "root" enhancer file. The "`inc/enhancer2.xml`" is an example of "including" another XML file in the "`inc`" subdirectory of the current folder of the `enhancer.xml` file.

The syntax of the `NativeEnhancerCmd` command is:

```
java com.innowhere.jnieasy.core.NativeEnhancerCmd urlClassDescXML
outputDir
```

Where *`urlClassDescXML`* is the URL of the XML root descriptor (or a specific class enhancer file) and *`outputDir`* is the root directory to write the enhanced classes (is highly recommended the same directory of the non enhanced `.class` files). But these is a very simplified command, the java interpreter must locate the .class archives (a `classpath` must be specified) and the JNIEasy DLL must be located (must be in the search path specified with the `java.library.path` property).

Is highly recommended to use Ant to perform the enhancement tasks. The Ant file `build.xml` provided with JNIEasy is a complete cross-platform example of Ant tasks, only a few variables of /conf/conf.properties file must be modified to setup a development environment of JNIEasy.

The XML descriptor root lists the classes to enhance, but a user defined native capable class being enhanced may need the native description of another user defined native class, if this user defined native class is not already enhanced (declared later in the list) the framework will search in the Java classpath to locate the XML enhancer descriptor file with the file name according the name rules and will enhance this class before (cyclic references are managed with no problem), more info in the On load enhancer section. If a class was already enhanced the enhancer does nothing (the .class file must be removed or the source class recompiled to re-enhance again a .class).

The content of the top most root file `enhancer.xml` is:

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- enhancer.xml -->

<jniEasyEnhancer version="1.1">
```

```
    <include file="manual/enhancer.xml" />
    <include file="win32exam/win32/enhancer.xml" />
</jniEasyEnhancer>
```

8.2 ON LOAD ENHANCER

The "on load enhancer" is an optional feature useful to avoid the filesystem enhancement of user defined native capable classes. The enhancement is performed in runtime using a special framework's `ClassLoader` when the class is used first time. If you need a quick loading of your program this feature must not be used because the enhancement is a bit slow task (use the filesystem enhancement on development time).

The `ClassLoader` enhancer enhances the original non-enhanced .class file in memory before submitting the bytecode to the JVM using the `ClassLoader.defineClass(String,byte[],int,int)` method. To do the enhancement task the enhancer `ClassLoader` needs to locate the XML enhancer descriptor file of the class, the search is performed with the method `ClassLoader.getResource(String)` using the name convention of XML enhancer files, the XML file must be in the classpath; the most simple approach is to put in the same folder the XML and the .class files.

Example of "on load enhancer" activation:

```
NativeEnhancer enhancer = JNIEasy.get().getEnhancer();

ClassLoader myParentLoader = Thread.currentThread().getContextClassLoader();

String[] included = new String[] { "examples.manual.*" };

String[] excluded = new String[] { StartInterface.class.getName() };

ClassLoader enhancerCL =

            enhancer.enableOnLoad(included,excluded,myParentLoader);

Class clsStart =

                Class.forName("examples.manual.Start",true, enhancerCL);

// Start class implements StartInterface

StartInterface start = (StartInterface)clsStart.newInstance();

start.run(); // method using native capable classes not enhanced
```

See `Enhancer.enableOnLoad(String[],String[],ClassLoader)` for more info.

9. JAVA CODE GENERATION

The Java code generation utilities are especially useful to generate the Java proxy code of methods (usually C methods) of DLLs exporting tons of methods. Usually a program uses a small group of methods of a big DLL, for example the Windows `User.dll`, the enhancer approach can spend very much time and memory initializing all proxy methods, used or not used in the program. The Java code generated by the generation utilities follows the principle: "the first use creates the proxy", there is no creation and registry of any internal proxy instance (using `DynamicLibrary` methods) until the method is first used, the second time will use the previously created proxy. The typical approach is group the C exported methods of a DLL in the same code generated Java class.

The generated methods are not synchronized to achieve the maximum performance, but they can work in a multithread environment because the creation of proxy instances is synchronized by the framework (the framework ensures the creation of a single proxy instance when two threads try in the same instant to create the same proxy instance).

In spite of the main objective is to generate C proxy methods, constructors and C++ proxy methods can be generated too, the generated Java class can be enhanced in a following step. This is another approach to create native capable classes with methods working as proxy of C/ C++ methods, these proxy methods with generated code do not need to be enhanced (they work already as proxies), use this technique if you are more comfortable with less code hidden by the enhancer.

The generation task is usually performed using the `NativeJavaCodeGeneratorCmd` class in the command line or Ant task, or using the method:

```
NativeJavaCodeGenerator.generate(String,String,String[],String[])
```

In both cases, the generation is made in "batch" mode using a "root" XML file listing the XML code generation descriptor files of classes to be generated. The name of this "root" XML file is not fixed, the XML generation descriptor of a class neither has a fixed name, but the following format is recommended:

```
SimpleClassName.jnieasy.jgen.xml
```

The XML file syntax is very similar to the enhancer counterpart.

The following example file (`proxygen.xml`) is an example of a XML root file:

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- proxygen.xml -->

<jniEasyJavaCodeGen version="1.1">

    <include file="MyCPPClassOnDLLGen.jnieasy.jgen.xml" />


    <include file="inc/proxygen2.xml" />

</jniEasyJavaCodeGen>
```

An example of a XML generation class descriptor file: it follows a similar syntax to the enhancer counterpart (using MSVC name mangling, the same example in the distribution is cross-compiler):

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- MyCPPClassOnDLLGen.jnieasy.jgen.xml -->
<jniEasyJavaCodeGen version="1.1">
  <fileGen name="MyCPPClassOnDLLGen"> <!-- generator adds .java -->
    <package name="examples.manual" >
      <imports>
        <import class="java.io.*" />
      </imports>
      <class name="MyCPPClassOnDLLGen" extends="Object"
                implements="Serializable,Cloneable"
                libraryPath="MyLibrary" >

        <field class="int" name="virtualTable" />
        <field class="double" name="value" />

        <freeCode>
    protected int virtualTable;
    protected double value;

    /* Empty constructor */
    MyCPPClassOnDLLGen()
    {
    }
     </freeCode>

        <constructor
           nativeName="?create@MyCPPClassOnDLL@@SGPAV1@HH@Z">
```

```
    <param class="int" name="a" />
    <param class="int" name="b" />
  </constructor>

  <method name="destroy" methodType="C"
     nativeName="?destroy@MyCPPClassOnDLL@@SGXPAV1@@Z">
    <return class="void" />
    <params>
      <param class="MyCPPClassOnDLL" name="obj" />
    </params>
  </method>

  <method name="addStatic" methodType="C"
     nativeName="?addStatic@MyCPPClassOnDLL@@SG_JHH@Z">
    <return class="long" />
    <params>
        <param class="int" name="a" />
        <param class="int" name="b" />
    </params>
  </method>

  <method name="sub" methodType="CPP"
     nativeName="?sub@MyCPPClassOnDLL@@UAGNHH@Z">
      <return class="double" />
      <params>
          <param class="int" name="a" />
          <param class="int" name="b" />
      </params>
  </method>
```

```
            <method name="varargsEx" methodType="C"
                    nativeName="?varargsEx@MyCPPClassOnDLL@@SAXPADZZ"
                callConv="c_call">
                <return class="void" />
                <params>
                    <param class="byte[]" name="buffer" />
                    <param dec="Object..." name="args" />
                </params>
            </method>
        </class>
    </package>
</fileGen>


</jniEasyJavaCodeGen>
```

Generates a class file, `MyCPPClassOnDLLGen.java`, very similar to the class `MyCPPClassOnDLL` seen before, with the exception of the methods now containing the necessary proxy code to call the C/C++ matched method.

Note the `name` and `class` attributes of fields and methods (returns) missing in the enhancer version (because the code generation constructs the Java class from scratch). The `extends` and `implements` attributes are declared to illustrate its use. The `import` (tag) declaration is needed to resolve the `Serialize` interface in the generated class.

If the Java class references other user defined native capable class, this must be previously enhanced or must be located the XML enhancer descriptor in the classpath (because the on load enhancer is used if necessary)

The syntax of the `NativeJavaCodeGeneratorCmd` command is:

```
java com.innowhere.jnieasy.core.NativeJavaCodeGeneratorCmd
    [-enhanceOnLoad includeImportList
    [-excludeImports excludeImportList]] xmlDescriptorPath  outputDir
```

The complete syntax is documented in the class `NativeJavaCodeGeneratorCmd`.

The distribution's `build.xml` contains Ant task examples to call the code generation tool.

The top most root is `proxygen.xml` defined as:

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- proxygen.xml -->

<jniEasyJavaCodeGen version="1.1">

    <include file="manual/proxygen.xml" />

    <include file="win32exam/proxygen.xml" />

</jniEasyJavaCodeGen>
```

The generated class `MyCPPClassOnDLL` must be enhanced later to work as a proxy of the `MyCPPClassOnDLL` C++ class, anyway, static methods can be called without enhancement (the main objective of the code generator, because most of the "big" DLLs are C libraries).

A partial result of the generated code:

```
package examples.manual;


import java.io.*;


public class MyCPPClassOnDLLGen extends Object

            implements Serializable,Cloneable

{

    public static com.innowhere.jnieasy.core.JNIEasy jnieasyRoot =

                com.innowhere.jnieasy.core.JNIEasy.get();

    public static com.innowhere.jnieasy.core.mem.DynamicLibrary

                jnieasyDLL =

                jnieasyRoot.getDLLManager().get("MyLibrary");

    public static com.innowhere.jnieasy.core.method.NativeBehavior[]

                jniEasyMethodList =

      new com.innowhere.jnieasy.core.method.NativeBehavior[ 4 ];

...

    public static long addStatic(int a,int b)

    {

        com.innowhere.jnieasy.core.method.NativeBehavior jnieasyMethod

                        = jnieasyMethodList[2];
```

```
            if (jnieasyMethod == null)
            {
                String jnieasyFuncName =
                                "?addStatic@MyCPPClassOnDLL@@SG_JHH@Z";
                jnieasyMethod = jnieasyMethodList[2] =
                                    jnieasyDLL.addCMethod(jnieasyFuncName,
                 jnieasyRoot.getTypeManager().dec(long.class).decVarType(),
                    new java.lang.Object[]
{jnieasyRoot.getTypeManager().dec(int.class).decVarType().decParameter(),
jnieasyRoot.getTypeManager().dec(int.class).decVarType().decParameter()},
                      com.innowhere.jnieasy.core.typedec.CallConv.STD_CALL
                    );
            }
            return
((com.innowhere.jnieasy.core.method.NativeStaticMethod)jnieasyMethod).
                  callLong(new java.lang.Object[]
                   {new java.lang.Integer(a),new java.lang.Integer(b)});
        }
. . .
}
```

A simple invocation example:

```
long res = MyCPPClassOnDLLGen.addStatic(1,2);
System.out.println("Must be 3: " + res);
```

The `varargsEx` method is interesting because contains a "varargs" parameter, in this example the `dec` attribute is used to illustrate the `"Object..."` declaration following the Java 5 notation; this expression declares the parameter as "varargs" with type `Object[]`. This declaration is equivalent to declare the parameter with `class="Object[]"` and `varargs="true"`.

10.DIRECT MEMORY MANIPULATION

JNIEasy offers two interfaces (with two default implementation classes) to manipulate the native memory: `NativeBuffer` and `NativeBufferIterator`.

A `NativeBuffer` object represents a contiguous native memory section and has a native address, using this object the native memory can be read and modified directly without native objects. The buffer can own the native memory or be attached, if the buffer owns the memory this is freed automatically when the object goes out the scope (garbage collected), but can be explicitly freed with the method `NativeBuffer.free()`. A buffer can be created with the methods:

- `NativeManager.allocateBuffer(long size)` to create a buffer allocating memory.
- `NativeManager.attachBuffer(long address,long size)` to attach the new buffer to an already allocated memory section. If the `size` is `-1` the buffer is considered unlimited and the upper bound is not checked.

The `NativeBuffer` interface has get and set methods to access native memory data, all of them use an `offset` parameter, this offset is the 0 based relative memory position in bytes where to read or write (to calculate the absolute position the offset is added to the buffer's address).

Example:

```
NativeBuffer buffer = JNIEasy.get().getNativeManager().allocateBuffer(1024);

long offset = 16;

double value = buffer.getDouble(offset);

value = value*2;

buffer.setDouble(offset,value);
```

Every native instance holds a `NativeBuffer` (referenced by the `NativeStateManager`), the method `NativeCapableUtil.getBuffer(Object)` returns this object. Is not recommended to free the buffer of a native instance with `NativeBuffer.free()`, free the native memory of a native instance with `NativeCapableUtil.free(Object)`.

The `NativeBufferIterator` interface is used to move along the buffer's native memory. Every reading or writing moves forward the internal iterator position the read or written bytes. A buffer iterator is created with the method `NativeBuffer.newBufferIterator()`, the iterator created is "attached" to the parent buffer. Several iterators can read concurrently (mono or multithread) the same buffer with no problem, concurrent multithread is possible but obviously dangerous (`NativeBuffer` and `NativeBufferIterator` objects are not thread synchronized).

Example:

```
NativeBufferIterator it = buffer.newBufferIterator();

for(int i = 0; i < 10; i++)

    it.setInt(i);
```

writes 10 contiguous integers (`jint`) in the buffer.

11.CROSS-PLATFORM OPTIONS

Main objective of JNIEasy's cross-platform options is simple: one Java code, multiple operating systems, compilers and native layouts[14]. The cross-platform in this context means cross-operating system, cross-compiler and cross-native layouts.

To achieve this objective JNIEasy offers several features:

- Platform dependent size of addresses
- Multiple native memory sizes and alignments of primitive types
- A C "inspired" macro based system to resolve sizes and names

11.1PLATFORM DEPENDENT SIZE OF ADDRESSES

Java primitive data types have fixed sizes; it was a very successful rule, because a pure Java program works exactly equal in all platforms. But this is a problem making Java and C/C++ integration, because all pointer types like `void*` and `long` data type[15] are platform dependent: in a 32 bit platform pointers and `long` size is 4 bytes (same as `int`), in a 64 bit environment pointers have 8 bytes and `long` has 8 bytes too in a LP64 model[16] (fortunately `int` maintains the same size, 32 bits). In a 32 bit platform the C/C++ `long/int/`*`anytype*` – Java `int` correspondence does not work fully in a 64 bit platform; the portable correlation must be C/C++ `jint` and Java `int` or C/C++ `jlong` and Java `long`, but is very unusual to see C/C++ legacy code using JNI data types.

One solution is to offer 32 bit and 64 bit versions of Java programs based in JNIEasy, the 32 bit version will use the Java `int` and the 64 bit version the `long` data type will be used to map the C/C++ `long`, in a LP64 platform, or to map an address/handler (usually handlers are pointers to undocumented structures).

The other solution is to use cross-platform addresses.

11.1.1 Built-in facilities

JNIEasy is ready to the 64 bit platforms (x86 AMD64/EM64T architectures).

JNIEasy uses the `long` type to deal with native addresses, memory sizes and offsets, in a 32 bit platform the addresses (and related memory sizes) are promoted to `long` normally (the four upper bytes are ever 0 if the integer value is positive or FFFFFFFF if negative) and cast to 32 bits with a normal cast operation (the signed value is ever unaltered). In a 64 bit platform no cast will be made.

---

[14] C/C++ typically use macros to construct multiple native layouts of the same construction (class, structure …)

[15] In LP64 data models

[16] LP64 data model is used by gcc 64 bit compilers (`long` size is 8 bytes), MSVC uses a LLP64 model (`long` remains 4 bytes) but the `LONG_PTR` data type is platform dependent. More info at: http://en.wikipedia.org/wiki/64-bit#64-bit_data_models and http://www.amd.com/us-en/assets/content_type/DownloadableAssets/dwamd_AMD64_PortApp.pdf

References (pointers) have a platform dependent variable size automatically: a Java native reference in a Java user defined native capable structure, or C++ class etc (e.g. a field with a native capable data type or "can be native" reference) occupies 32 bits in a 32 bit platform and 64 bits in a 64 bit platform.

11.1.2 Cross-platform long

Another facility is to use the Java `long` data type as a cross-platform data type in a native perspective.

A cross-platform Java `long` occupies in the native memory 32 bits in a 32 bit platform and 64 bits in a 64 bit platform. A cross-platform `long` works as the C/C++ `long` and pointers types, using the cross-platform `long` the 64 bits and 32 bits Java versions are the same, only one API.

To declare a cross-platform `long` must be used the XML attribute "`address`", if set to "`true`" the `long` native size is platform dependent (the size of addresses of the platform). Arrays, `Long`, `NativeLong` and `NativeLongObject` data types can contain a `long` cross-platform, declaring the `long` component as address with the optional `<component>` child element (see Primitive wrapper types and Primitive object types).

To declare a cross-platform `long` using a programmatic manner use the `NativeTypeManager.decAddress()` method.

Examples:

```
public class MyStructure

{

. . .

    protected long crossPlatLong;

    protected long[] crossPlatLongArray = new long[]{1,2};

    protected Long crossPlatLongObject = new Long();

    protected NativeLong crossPlatNativeLong;

    protected NativeLongObject crossPlatNativeLongObject;


    public MyStructure()

    {

     . . .

      NativeTypeManager typeMgr = JNIEasy.get().getTypeManager();

      this.crossPlatNativeLong = (NativeLong)typeMgr.decAddress().

            decObjectWrapper(NativeLong.class).newValue();

      this.crossPlatNativeLongObject = (NativeLongObject)typeMgr.decAddress().
```

```
                decObjectWrapper(NativeLongObject.class).newValue();
        }
    ...
    }
```

XML enhancer descriptor:

```
    . . .
    <field name="crossPlatLong" address="true" />
    <field name="crossPlatLongArray" >
       <component address="true" />
    </field>
    <field name="crossPlatLongObject" >
       <component address="true" />
    </field>
    <field name="crossPlatNativeLong" >
       <component address="true" />
    </field>
    <field name="crossPlatNativeLongObject" >
       <component address="true" />
    </field>
    . . .
```

In a 32 bit platform the size of the `crossPlatLong` attribute is 32 bits, the `crossPlatLongArray` reference points to an array containing 32 bit integers and `crossPlatLongObject`, `crossPlatNativeLong` and `crossPlatNativeLongObject` references point to 32 bit integers.

11.2 MULTIPLE NATIVE MEMORY SIZES AND ALIGNMENTS OF PRIMITIVE TYPES

Java primitive types (`boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, `double`) have a default native type, the native type can be seen as the JNI type (`jboolean`, `jbyte`, `jchar`, `jshort`, `jint`, `jlong`, `jfloat`, `jdouble`). Java ensures fixed memory sizes to the primitive types in the Java side, by default native counterparts in JNIEasy have the same sizes (and alignments), but this can be changed on user demand.

Custom sizes help the developer to program in Java with more fidelity to the native side. Example: the native counterpart to the Java `char` is not the C `char`, it is `short` or `wchar_t` (with 2 bytes) and vice versa, the C `char` counterpart is the Java `byte`. The Java `byte` conventions are not the same as the Java `char`, if contained data is a character the `byte` data type is not the best match. Another example is Java `boolean` (1 byte) and `BOOLEAN` Win32's macro (both types conceptually match but the size does not).

JNIEasy lets to change the default size and alignment of the native primitive type layout of a Java primitive data type globally or with a per-element basis (usually primitive fields or method parameters).

11.2.1 Changing the global default size and alignment

The methods `NativeTypeManager.setDefaultXSize(long size, long alignment)`, where `X` is `Boolean`, `Char`, `Int` etc, can be used to define the default size and alignment of primitive types globally. To set the desired default values before any native memory access is highly recommended. Usually the alignment is the same as the memory size, the methods `NativeTypeManager.setDefaultXSize(long size)` set the alignment to the memory size passed as argument.

Example:

```
JNIEasy.get().getTypeManager().setDefaultCharSize(1,1);
```

sets the default size and alignment of Java `char` in native as 1 (like the C `char`).

To know the valid values see the javadoc documentation of `setDefaultXSize(long,long)` methods.

`NativeBuffer` and `NativeBufferIterator` methods are not affected by the primitive default sizes, the native sizes are ever the Java sizes.

11.2.2 Changing the default size and alignment of a native element

Any primitive type element can be declared with a custom native memory size and/or alignment with the XML attributes `memSize` and `prefAlignSize`.

Example:

```
public class MyStructure
{
...
    protected char cChar;
...
}
```

and XML enhancer descriptor:

```
. . .
<field name="cChar" memSize="1" />
```

```
. . .
```

Previous XML code declares `cChar` with native memory size 1 byte and alignment 1 byte (if memory size is specified and alignment is not, the alignment is set to the memory size).

"Verification" with Java:

```
NativeTypeManager typeMgr = JNIEasy.get().getTypeManager();

StructureDescriptor strucDesc =

      (StructureDescriptor)typeMgr.getClassDescriptor(MyStructure.class);

NativeFieldDescriptor fieldDesc = strucDesc.getField("cChar");

TypeNative fieldType = fieldDesc.getVarType().getType();

System.out.println("Must be 1: " + fieldType.size());

System.out.println("Must be 1: " + fieldType.preferredAlignSize());
```

Using a programmatic approach the method `NativeTypeManager.decPrimitive(Class clasz,long size, long alignment)` declares a primitive native type with the memory size and alignment specified (using the method `NativeTypeManager.decPrimitive(Class clasz,long size)` the alignment is set as the specified size).

11.2.2.1 Phantom primitive fields

Size 0 is a valid native memory size and alignment of primitive types. A primitive field with native memory size 0 is a *phantom field*; a phantom field has no native memory layout influence (alignment must be set to 0 too). They are interesting when a specific platform declares a field not declared in another platform using C macros. This feature is only useful used in conjunction with the JNIEasy's macro system.

11.3C "INSPIRED" MACRO SYSTEM TO RESOLVE SIZES AND NAMES

The C macro system is "a blessing" in the C/C++ world because it enables the developer to face the incompatibilities of platforms, compilers, library versions …

Java has not a C macro like system because the `ClassLoader` mechanism and the WORA principle (the JVM works the same on every platform) are the alternatives. But in JNIEasy these features are not sufficient because JNIEasy needs to deal with native elements, a native element can be different between platforms. To face this problem JNIEasy introduces an C "inspired" macro system: a name or size can vary between platforms, the used name or size is resolved on runtime depending on declared macros, defined or not, in runtime.

Main identified cases:

1. DLL names: a dynamic library name can change between platforms. Example: `MSVCRT.DLL` (lib name, MSVCRT) is the C runtime library on Windows, `libc.so.6` on Linux, `libc.so.1` on Solaris and `libc.dylib` on Mac OS X.

2. Exported symbols/names of methods and fields: the name mangling is different between compilers, it affects to C++ exported elements and standard call C methods (MSVC add a @N suffix).

3. Size (and alignment) of primitive data types: primitive data types (and derivatives) can have different size between platforms (and compilers), for instance the `long` data type is 4 bytes in gcc Linux/Mac/Solaris x86 and 8 bytes in x86_64, the `LONG_PTR` has variable size too in Windows-MSVC.

JNIEasy provides a macro name/value registry with methods like `NativeTypeManager.defineMacro(String,Object)`, `NativeTypeManager.getMacro(String)` etc. These macros are used to resolve expressions where the selected value is obtained if a concrete macro was declared. The approach is similar to the `#ifdef` *`MACRO`* or `#if defined(`*`MACRO`*`)` C/C++ preprocessor instructions.

Three methods can be used to parse macro based expressions:

1. `NativeTypeManager.`**`parseTextWithMacros`**`(String textExpr)`

   Returns the text prefixed with a previously defined macro (see the javadoc for a full syntax description).

   Example:

   ```
   "WIN32:MSVCRT;Linux:/lib/libc.so.
   6;Mac:/usr/lib/libc.dylib;SunOS:/lib/libc.so.1;OtherLibc"
   ```

   The name "`MSVCRT`" is returned if the "`WIN32`" macro was defined, else "`/lib/libc.so.6`" if the "`Linux`" macro was defined, else "`/usr/lib/libc.dylib`" if the "`Mac`" macro was defined else "`/lib/libc.so.1`" if the "`SunOS`" macro was defined else "`OtherLibc`".

2. `NativeTypeManager.`**`parseNameWithMacros`**`(String nameExpr)`

   Is used to resolve names prefixed with macros, is based on `parseTextWithMacros` (basically trims the leading and trailing spaces).

3. `NativeTypeManager.`**`parseMemorySizeWithMacros`**`(String sizeExpr)`

   Returns the size prefixed with a previously defined macro (see the javadoc for a full syntax description).

   Example: "`MSC:4;gcc:8;REGISTER_SIZE`" the size returned is 4 if the "`MSC`" macro was defined, else is 8 if the "`gcc`" macro was defined, else the value of the "`REGISTER_SIZE`" macro is used (must be defined with an integer value or an exception is thrown).

Several JNIEasy methods already support directly this macro based syntax like `DLLManager.get(String)/find(String)/remove(String)`, `DynamicLibrary.getAddress(String)/addDLLBehavior(String,…)/findBehaviorsByName(String)/findBehaviorByName(String,…)/removeBehaviorsByName(String)/removeBehaviorByName(String,…)`, `JNIEasyLibrary.exportBehavior(String,…)`, `NativeTypeManager.decPrimitive(Class,String,String)/decPrimitive(Class,String)/decString(Class,String)/decStringBuffer(Class,String)`.

XML attributes (enhancer and code generation) support this syntax too:

- `libraryPath` : parsed with `NativeTypeManager.parseNameWithMacros(String)`
- `nativeName` : parsed with `NativeTypeManager.parseNameWithMacros(String)`
- `encoding` : parsed with `NativeTypeManager.parseNameWithMacros(String)`
- `memSize` : parsed with `NativeTypeManager.parseMemorySizeWithMacros(String)`
- `prefAlignSize` : parsed with `NativeTypeManager.parseMemorySizeWithMacros(String)`
- `alignSize` : parsed with `NativeTypeManager.parseMemorySizeWithMacros(String)`

Example:

```
NativeTypeManager typeMgr = JNIEasy.get().getTypeManager();

String osname = System.getProperty("os.name");

if (osname.startsWith("Windows"))

  typeMgr.defineMacro("Windows");

else if (osname.startsWith("Linux"))

    typeMgr.defineMacro("Linux");

else if (osname.startsWith("Mac OS X"))

  typeMgr.defineMacro("MacOSX");

else if (osname.startsWith("SunOS"))

    typeMgr.defineMacro("SunOS");


String dllName =  "Windows:MSVCRT;Linux:/lib/libc.so.
6;MacOSX:/usr/lib/libc.dylib;SunOS:/lib/libc.so.1";

DLLManager dllMgr = JNIEasy.get().getDLLManager();

DynamicLibrary dll = dllMgr.get(dllName);

CMethod method = dll.addCMethod("abs",int.class,

            new Object[]{int.class},CallConv.C_CALL);

int res = method.callInt(new Object[]{ new Integer(-10) });

System.out.println("Must be true: " + (res == 10));
```

12.NATIVE TRANSACTIONS

JNIEasy supports, optionally, transactional manipulations of native memory in a similar fashion as JDO or Hibernate. In native transactions the native memory must be seen as a data repository; during a native transaction the original state of the native memory is saved before any modification and is restored to the original values if the transaction is aborted (rollback).

A native transaction has boundaries (begin and end) and is thread isolated (transactions are not shared by threads). The thread isolation is the only one isolation offered by a native transaction, the same native memory fragment can be modified concurrently by two threads with one or two live transactions (JNIEasy avoids the concurrent modification but there is no thread lock between transactions); every transaction keeps track of the memory modifications performed by the associated thread only. If the user is going to modify the same native instance concurrently in a transactional environment is recommended to make a global lock during every transaction.

Native transactions only work keeping the state of fields from native capable classes; direct native memory manipulations with `NativeBuffer` and native modifications performed in the native side are not monitored and are not transactional.

Rollback behavior

a) The native instance was made native inside the transaction: the allocated memory is freed.

b) The native instance was modified inside the transaction: the modified native memory is returned to the previous state to the modification.

c) The native instance was freed (non-native) inside the transaction: the instance is made native again and the values of the native memory are restored.

The transaction object implementing the interface `NativeTransaction`, associated to the current (calling) thread can be obtained with the method:

```
NativeManager.currentTransaction()
```

A transaction begins with the method `NativeTransaction.begin()` and ends with the methods `NativeTransaction.commit()` and `NativeTransaction.rollback()`.

Example:

```
NativeManager nativeMgr = JNIEasy.get().getNativeManager();


MyStructure structUpdatedInside = new MyStructure();

structUpdatedInside.setIntArray4(new int[]{1,2});

nativeMgr.makeNative(structUpdatedInside);


MyStructure structNewInside = new MyStructure();
```

```
structNewInside.setIntArray4(new int[]{1,2});


MyStructure structFreedInside = new MyStructure();
nativeMgr.makeNative(structFreedInside);
structFreedInside.setIntArray4(new int[]{1,2});


NativeTransaction txn = nativeMgr.currentTransaction();


txn.begin();


nativeMgr.makeNative(structNewInside);
structNewInside.setIntArray4(new int[]{3,4});


nativeMgr.free(structFreedInside);
structFreedInside.setIntArray4(new int[]{3,4});


structUpdatedInside.setIntArray4(new int[]{3,4});


txn.rollback();


boolean res;
int[] array;


res = NativeCapableUtil.isNative(structNewInside);
System.out.println("Must be false: " + res);
array = structNewInside.getIntArray4();
System.out.println("Must be 1: " + array[0]);
```

```
res = NativeCapableUtil.isNative(structFreedInside);

System.out.println("Must be true: " + res);

array = structFreedInside.getIntArray4();

System.out.println("Must be 1: " + array[0]);


array = structUpdatedInside.getIntArray4();

System.out.println("Must be 1: " + array[0]);
```

13.LISTENERS

JNIEasy provides an event notification system monitoring the lifecycle of native instances.

The user can register listeners to receive these events; a listener is an instance of a class implementing a listener interface and registered, if necessary, as listener in JNIEasy.

There are two types of listener interfaces:

13.1INTERFACES TO BE IMPLEMENTED BY USER DEFINED NATIVE CAPABLE CLASSES

The native instance changing its native state receives the matched event according the interface(s) implemented; the instance works itself as a listener and does not need to register. The interface names have the following pattern: `EventTypeCallback`.

Interface	Related NativeManager method	Methods	Notes
AttachCallback	`attach(Object,NativeAddress,long)`	Pre Post	Is called pre and post the attachment. The "pre" method is not enhanced.
AttachCopyCallback	`attachCopy(Object,Object)`	Pre Post	Is called pre and post copy to native instance a non-native instance (copy)
DetachCallback	`detach(Object,int,boolean)`	Pre Post	Is called pre and post detachment the native instance. The "post" method is not enhanced.
DetachCopyCallback	`detachCopy(Object)`	Pre Post	Is called pre and post when creating a detached clone of the native instance.
FetchCallback	`fetch(Object,int)`	Post	Is called post fetching the instance from native memory. This method is not enhanced, useful to synchronize non-native fields with native fields.
MakeNativeCallback	`makeNative(Object)` `makeNative(Object,NativeAddress,long)`	Pre (two modes) Post	Is called pre and post made native (writing the native memory). The "pre" methods are not enhanced, are useful to prepare native fields before to write to native memory
UnFetchCallback	`unFetch(Object,int)`	Pre	Is called pre "unfetching" the

			instance to native memory. This method is not enhanced, useful to prepare native fields before to write to native memory.

Example: modifying `MyStructure`

```
public class MyStructure implements MakeNativeCallback
{
    public void jnieasyPreMakeNative(NativeAddress nativeAddress,long offset)
    {
        System.out.println("PreMakeNative (attached)");
        System.out.println("Must be false: " +
                           NativeCapableUtil.isNative(this));
    }

    public void jnieasyPreMakeNative()
    {
        System.out.println("PreMakeNative (memory owner)");
        System.out.println("Must be false: " +
                           NativeCapableUtil.isNative(this));
    }

    public void jnieasyPostMakeNative()
    {
        System.out.println("PostMakeNative");
        System.out.println("Must be true: " +
                           NativeCapableUtil.isNative(this));
    }
}

NativeManager natMgr = JNIEasy.get().getNativeManager();
```

```
MyStructure obj = new MyStructure();

natMgr.makeNative(obj); // Calls jnieasyPreMakeNative()
                        // and jnieasyPostMakeNative()
NativeBuffer address = NativeCapableUtil.getBuffer(obj);


MyStructure obj2 = new MyStructure();

natMgr.makeNative(obj2,address,0);
                    // Calls jnieasyPreMakeNative(Address,long)
                    // and jnieasyPostMakeNative()
```

13.2 INTERFACES IMPLEMENTED BY NORMAL CLASSES WORKING AS LISTENERS

Instances of these classes must be registered in JNIEasy to receive events. These are the `InstanceLifecycleListener` based interfaces; listener methods will receive `InstanceLifecycleEvent` events. The listener interfaces are symmetric to the native instance interfaces.

A class working as a listener must implement only one `InstanceLifecycleListener` interface.

Interface	Related symmetric instance interface	Event class
AttachLifecycleListener	AttachCallback	AttachLifecycleEvent
AttachCopyLifecycleListener	AttachCopyCallback	AttachCopyLifecycleEvent
DetachLifecycleListener	DetachCallback	DetachLifecycleEvent
DetachCopyLifecycleListener	DetachCopyCallback	DetachCopyLifecycleEvent
FetchLifecycleListener	FetchCallback	FetchLifecycleEvent
MakeNativeLifecycleListener	MakeNativeCallback	MakeNativeLifecycleEvent
UnFetchLifecycleListener	UnFetchCallback	UnFetchLifecycleEvent

Example:

```
package examples.manual;
```

```
import com.innowhere.jnieasy.core.util.NativeCapableUtil;
import com.innowhere.jnieasy.core.listener.MakeNativeLifecycleEvent;
import com.innowhere.jnieasy.core.listener.MakeNativeLifecycleListener;


public class MakeNativeListenerExample
       implements MakeNativeLifecycleListener
{
    public MakeNativeListenerExample()
    {
    }


    public void preMakeNative(MakeNativeLifecycleEvent event)
    {
        if (event.isAttaching())
            System.out.println("PreMakeNative (attached)");
        else
            System.out.println("PreMakeNative (memory owner)");
        System.out.println("Must be false: " +
                NativeCapableUtil.isNative(event.getSource()));
    }


    public void postMakeNative(MakeNativeLifecycleEvent event)
    {
        System.out.println("PostMakeNative");
        System.out.println("Must be true: " +
                NativeCapableUtil.isNative(event.getSource()));
    }
}
```

When register a listener, the classes "interested" must be specified:

```
MakeNativeListenerExample listener = new MakeNativeListenerExample();
JNIEasy.get().getNativeManager().addInstanceLifecycleListener(listener,

  new Class[]{MyStructure.class});
```

the listener will receive "make native" events of `MyStructure` objects. Now the previous "makeNative" examples can be executed again to "fire" events.

```
NativeManager natMgr = JNIEasy.get().getNativeManager();

MyStructure obj = new MyStructure();

natMgr.makeNative(obj); // Calls preMakeNative() and postMakeNative()

NativeBuffer address = NativeCapableUtil.getBuffer(obj);


MyStructure obj2 = new MyStructure();

natMgr.makeNative(obj2,address,0);

  // Calls preMakeNative() and postMakeNative()
```

14.MULTITHREADING SUPPORT

JNIEasy has been designed to run in a multithread environment.

Key parts of the JNIEasy framework are multithread safe, sometimes the multithread support is achieved using thread locking but only when necessary, for instance, native method calls and Java callback calls are not locked to achieve the maximum parallelism.

There are parts or uses of the framework not synchronized, consciously, for instance the `NativeBuffer` methods: concurrent modification of the same native memory is a risky task; the developer must synchronize concurrent access to shared native memory.

The operations of synchronization between Java fields and native memory in native classes are synchronized in a per native instance basis; Java fields can be read/modified concurrently with no problem from a framework's point of view, but this is not recommended (specially concurrent modification) from an application's point of view, developer is encouraged to synchronize concurrent access to shared native instances and native instances sharing the same native memory (several native instances can be attached to the same native memory part).

Native memory access performed from native side is in no way synchronized.

In brief, no developer synchronization code is necessary if concurrent threads do not share the same native memory parts.

15.SERIALIZATION

Native capable classes (user and predefined) are "serializable aware".

Every native capable class has two internal non-static fields: `jnieasyType` and `jnieasyNativeStateManager`, both fields are declared as transient, and not serialized by default, this ensures the "native state" is not serialized. Any other field holds data and must be serialized.

All predefined native classes implement `Serializable` and `Cloneable` interfaces, user defined arrays and pointers implement these interfaces too by default, user defined Java structures, C ++ classes and unions do not, the developer can do it explicitly.

The serialized map of a user defined class not enhanced and enhanced is the same but the serial UID is different, to avoid this, a `serialVersionUID` field must be added, for instance:

```
private static final long serialVersionUID = 1L;
```

this ensures that a serialized enhanced class instance can be deserialized as a not enhanced class instance and vice versa.

The serialization compatibility between enhanced and not enhanced classes is very important to pass objects in a distributed environment, a peer can use the enhanced version of the class and the other peer can use the non-enhanced version.

A native instance can be serialized and distributed, of course the deserialized instance is non-native, and can be again made native if the peer works with JNIEasy (using the `attach` and `makeNative` methods). There is a problem: serialization uses Java reflection, Java reflection is not controlled by the enhancer, if an instance is going to be serialized and the serialized instance must be a copy of the current native memory state, a deep "unfetch" must be performed before the serialization process.

Another option is to serialize non-native instances; of course a native instance can be freed or detached, but to keep the native object tree untouched there are two `NativeManager` methods to use a detached cloned object tree instead the "live" objects:

1) `Object detachCopy(Object obj)`

Creates a "deep" detached clone of the native object argument and tree (native fields). The native relations are cloned too (the detached tree is independent from the original) including circular relationships (direct and indirect).

2) `void attachCopy(Object nativeObj,Object detachedObj)`

Copy the content of a non-native object to a native object (both objects must have the same type). The non-native object is kept as is, but member native capable objects are made native and member of the native object.

Using these methods an entire native tree can be cloned, serialized, distributed, unserialized and copied to a symmetric native tree.

Example:

```
public class MyStructure
                implements MakeNativeCallback,Serializable,Cloneable
```

```
{
...
}

MyStructure struct = new MyStructure();
JNIEasy.get().getNativeManager().makeNative(struct);

MyStructure struct2 = new MyStructure();
JNIEasy.get().getNativeManager().makeNative(struct2);

MyStructure structCopy =
      (MyStructure)JNIEasy.get().getNativeManager().detachCopy(struct);

ByteArrayOutputStream repOut = new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream(repOut);
oos.writeObject(struct);
oos.close();

ByteArrayInputStream repIn =
            new ByteArrayInputStream(repOut.toByteArray());
ObjectInputStream ois = new ObjectInputStream(repIn);
MyStructure structCopy2 = (MyStructure)ois.readObject();
ois.close();

System.out.println("Must be true: " + (struct2.getIntArray3() != null));
structCopy2.setIntArray3(null);
JNIEasy.get().getNativeManager().attachCopy(struct2,structCopy2);
System.out.println("Must be null: " + struct2.getIntArray3());
```

16.XML DESCRIPTORS REFERENCE

The XML descriptor structure of enhancer and Java code generation are specified in the XMLSchema format.

Due to performance reasons, JNIEasy does not validate a XML file against its schema, wherefore is not necessary to declare the XMLSchema in the XML header, but is recommended in development time to avoid mistakes. Many tools like XML tools and Java IDEs validate XML files against its XMLSchema.

The XMLSchema specification has limitations, this chapter explains the precise information missing in the XMLSchemas and a functional meaning of elements and attributes.

16.1XML ENHANCER DESCRIPTOR REFERENCE

The XML enhancer descriptor structure is specified in the XMLSchema archive:

`JNIEasy.enh.xsd`

A header example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Archive MyStructure.jnieasy.enh.xml -->
<jniEasyEnhancer version="1.1"
    xmlns="http://www.innowhere.com/jnieasy/enhancer"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.innowhere.com/jnieasy/enhancer
        ../../../schemas/JNIEasy.enh.xsd">

    <package name="examples.manual">
        <imports />
        <class name="MyStructure" type="structure">
...
```

16.1.1 Element: jniEasyEnhancer

The root element.

- `version` attribute : is mandatory, current version is 1.1.

16.1.2 Element: include

Used to include other XML files. Useful to enhance classes from command line in "batch" mode.

- `file` attribute : specifies the file to include, can be specified as complete URL or a relative file path. Is required.

16.1.3 Element: package

Declares the package of the contained classes (usually one class).

Attributes:

- `name`: the format is the same as the Java `package` declaration. Is required.

Child nodes: `<imports>` and `<class>`.

16.1.4 Element: imports

Parent element of `<import>` declarations.

16.1.5 Element: import

Declares a class pattern to resolve relative class names used in the following `<class>` elements.

Attributes:

- `class`: the format is the same as the Java `import` declaration. Is required

The `"java.lang.*"` import is implicitly declared following the Java rule; for instance, the `"String"` class name will be resolved as `"java.lang.String"`.

The package name is used too as a prefix to resolve class names without package (simple class names).

16.1.6 Element: class

Declares a user defined native capable class to be enhanced.

Attributes:

- `name`: declares the "simple" name of the class. The format is the same as the name of a Java `class` declaration. Is required.

- `type`: declares the native element this Java class represents. Possible values are: `structure`, `class`, `union`, `array`, `pointer` or `callback`. Is required.

- `allFields`: tells to the enhancer if all fields must be or not must be enhanced, if `true` the enhancer tries to enhance all native capable non-transient member fields although the field is not declared in the XML. Is optional, the default value is `true`.

- `allMethods`: tells to the enhancer if all methods must be or not must be enhanced, if `true` the enhancer tries to enhance all native capable methods although the method is

not declared in the XML. It is only applied to "multiple field containers" like classes working as structures, C++ classes or unions. Is optional, the default value is `false`.

- `libraryPath`: specifies the name or path of the Dynamic Link Library to be used in this class, if some method is declared as a proxy of a native method. It is only applied to "multiple field container classes" (structures, C++ classes or unions). The library path/name follows the conventions of `System.loadLibrary(String)` and can be cross-platform using macros, the full format is explained in the javadoc of `DLLManager.get(String)`.

- `alignSize`: specifies the alignment size to be applied to this class. Is optional. Allows a cross-platform syntax using macros.

Child nodes: `<field>`, `<constructor>`, `<method>`, `<fieldMethod>`.

16.1.7 Element: field

Declares a field as native and describes its native representation.

Attributes:

- `name`: the name of the field to be native. Is required.

- `enhance`: specifies if the field must be native or keep as non-native. The main use is to exclude (value `false`) a file to be enhanced. Is optional, the default value is `true`.

- `alignSize`: specifies the alignment size to be applied to this field. Is optional and only is applied to fields of "multiple field container classes" (structures, C++ classes or unions). Allows a cross-platform syntax using macros.

- `union`: specifies the begin and end of a group of fields forming an anonymous union. Values `begin` and `end`. Is optional and only is applied to fields of "multiple field container classes" (structures, C++ classes or unions).

- Native variable type attributes: declares the native representation of the field type. Seen later in "Native variable type attributes and nodes".

Child nodes: nodes corresponding to the native variable type. Seen later.

16.1.8 Element: constructor

Declares a constructor as native and describes its native representation.

Attributes:

- `enhance`: idem `<field>`. If `false` the `<param>` child nodes and its `class` attributes specify the concrete constructor.

- `useReflection`: if `true` the constructor working as a callback is called using Java reflection, else is called with a direct call. Is optional, by default is `false`. Only is processed this attribute if `onLibrary` is not defined or `false` (the constructor works as a callback).

- `exportMethod`: if `true` the constructor is exported and found using the C method `findExportedMethodAddress(const char*)`. Is optional, by default is `false`.

- `onLibrary`: if `true` the constructor is a proxy of a "native constructor". Is optional, by default is `false`.
- `nativeName`: specifies the exported name in the DLL to link the constructor working as a proxy. The name must be the exported name or a cross-platform name, the full format is explained in the javadoc of the `NativeTypeManager.parseNameWithMacros(String)` method. Is ignored if `onLibrary` is not defined or `false` else is required.
- Common native behavior attributes: seen in "Common native behavior attributes".

Child nodes: `<param>` nodes specify the concrete constructor and native representation of parameters. Seen later.

16.1.9 Element: method

Declares a method as native and describes its native representation.

Attributes:

- `name`: the name of the method to be native. Is required.
- `enhance`: idem `<constructor>`.
- `useReflection`: idem `<constructor>`.
- `exportMethod`: idem `<constructor>`.
- `onLibrary`: idem `<constructor>`.
- `nativeName`: idem `<constructor>`, but in this case is optional, if not defined the native name used is the method name.
- Common native behavior attributes: seen in "Common native behavior attributes".

Child nodes: `<return>` and `<params>` nodes specify the concrete method and native representation of return type and parameters. Seen later.

16.1.10 Element: fieldMethod

Declares a field seen as a native method (a field-method) and describes its native representation.

Attributes:

- `name`: the name of the field to be native as a field-method. Is required.
- `enhance`: idem `<constructor>`.
- `useReflection`: idem `<constructor>`.
- `exportMethod`: idem `<constructor>`.
- `onLibrary`: idem `<constructor>`.

- `nativeName`: idem `<method>`, but in this case the native name is the exported name of the field in the DLL, if not defined the native name used is the field name.

- Common native behavior attributes: seen in "Common native behavior attributes".

Child node: `<fieldType>` node specify the native declaration of the return and "value" parameter of the field-method. Seen later.

16.1.11 Common native behavior attributes

The method based (behaviors) elements: `<constructor>`, `<method>`, `<fieldMethod>` and native signature declarations can use the following attributes:

- `callConv`: specifies the native call convention used to call this method from native. Permited values are `std_call` and `c_call`. Is optional, by default is `std_call`.

- `thisClass`: specifies the "container" class of the method. It is required in user native capable callbacks to specify the "real" class containing the method if different to the user native capable callback class. In native signatures specifies the container class in constructors, instance methods and instance fields.

- `decSignature`: declares the native representation of the method signature with a string expression, the format of the expression is explained in the javadoc of the method `NativeSignatureManager.decBehavior(String,String[])`. This attribute is an alternative native signature declaration to use the `callConv`, `thisClass` attributes and the child nodes `<params>`, `<param>`, `<return>` and `<fieldType>`, these can be omitted.

- `params`: declares the list of classes of the constructor (a comma separated list of class names), method or field method to identify which one is declared. This attribute is an alternative to `<params>`, the default native representation of every declared Java data type is used. Elements `<return>` and `<params>` can be avoided, anyway `<return>` may be used to specify the native representation of the return data type beyond the default.

16.1.12 Element: return

Declares the native variable type of a method return and can be used in methods (not constructors, or field-methods). The attributes and child nodes are described later in "Native variable type attributes and nodes". If omitted the native variable type is the default of the Java type or the specified in the `decSignature` attribute.

16.1.13 Element: params

Is the node container of `<param>` nodes describing the native signature of a method (constructors do not need this container node). The `<params>` node follows the `<return>` node.

Optionally the `params` attribute, with a comma separated list of Java class names as value, may be used to avoid the `<param>` nodes. The default native representation of every declared Java data type is used.

16.1.14 Element: param

Declares the native variable type of a method parameter and can be used in methods and constructors (not field-methods). If omitted the native variable type is the default of the Java type or the specified in the `decSignature` attribute.

- `varargs`: specifies if the parameter is a variable number of arguments parameter. The parameter data type must be a Java array (`Object[]`, `String[]`, `NativeString[]`, `char[]` etc). A varargs attribute mimics in Java the C "…" parameter convention. Without this attribute set to true, the default convention of the array parameter type is to pass a pointer-to-array to the method, with the `varargs` attribute set to true all array elements are passed as parameters to the method with their "intrinsic/default" native type. In this context the default native type of Java primitive wrappers like `Integer`, `Character`, `Double` etc, is `int`, `char`, `double` etc (in other contexts the default native type of `Integer` is `int*` and so on). Is optional, by default is `false`.

Other attributes and child nodes are described later in "Native variable type attributes and nodes". The `name` attribute can be specified but is ignored by the enhancer.

16.1.15 Element: fieldType

Declares the native variable type of the field type of a field-method, this declaration is used to the return and "value" parameter of the field-method. Can be used only in field-methods. The attributes and child nodes are described later in "Native variable type attributes and nodes". If omitted the native variable type is the default of the Java type or the specified in the `decSignature` attribute.

16.1.16 Native variable type attributes and nodes

The nodes: `<field>`, `<return>`, `<param>`, `<fieldType>`, `<component>` and `<pointer>` can specify the native variable type description of the Java type involved (the type of the field, method return, parameter, field in a field-method), the fields and child nodes describing the native variable type are specified in the same way.

There are attributes shared by all types and attributes and child nodes related to a specific Java type or type group.

Common attributes:

- `varConv`: specifies if the native type is declared "by value" or "by pointer". Permited values are `byValue`, `byPtr` and `byDefault`. Is optional, the default value is the default convention of the native type.

- `class`: specifies the Java class of the native type, the value must be a class name (ex. `java.lang.String`) or a simple class name (`String`); if used a simple class name, the declared imports and the declared package are used to resolve the class name. Is optional if the Java type can be deduced, for instance in the `<field>` node the class is the field's Java type; but with `<param>` is needed to resolve the concrete Java method or to specify the parameter type of a unknown method signature being declared; with `<component>` or `<fieldType>` or `<pointer>` can be used to complete the declaration of a native generic interface or "can be native" type like `NativeCapableArray` or `java.lang.reflect.Field` (or `NativeFieldMethodReflection`) or `NativePointer` respectively.

  Example:

```
private NativePointer ptrToPtr;
```

The XML descriptor:

```
<field name="ptrToPtr2" >

    <pointer class="MyStructure" />

</field>
```

The `class` attribute in this case complete the native type declaration of `NativePointer`, the field `ptrToPtr` only points to `NativePointer` objects pointing (the internal pointer) to `MyStructure` objects.

- `dec`: declares the native variable type using a string expression, the format of the expression is explained in the javadoc of the method `NativeVarTypeManager.dec(String,String[])`. This attribute is an alternative declaration to use `varConv`, `class` attributes and the native type specific child nodes, these can be omitted. A special case is the `"…"` syntax only useful declaring a parameter type (`<param>` node): `dec="NativeType…"` declares a `NativeType` Java array (`NativeType[]`) as a varargs parameter; examples: `"Object…"`, `"String…"`, `"NativeString…"`, `"int…"` etc, more info see the `varargs` attribute of Element: param

16.1.17 String based native variable type

String based types like `String`, `StringBuffer`, `NativeString` and `NativeStringBuffer` can specify the encoding/size of the string with the attribute:

- `encoding`: declares the encoding/size of the string. Values are `ansi` and `unicode`. Supports a macro base expression. Is optional, default value is `ansi`.

16.1.18 Primitive types

The native view of primitive types is self described: the native memory size is by default the memory used by Java. This can be changed in a per-declaration basis.

- `memSize`: declares the native memory size, valid values are specified in the javadoc of the specific method `setDefaultXSize(long,long)` where `X` is `Boolean`, `Integer` etc. Is optional, default value is the Java size of the primitive type. Allows a cross-platform syntax using macros.

- `prefAlignSize`: declares the native memory alignment size, valid values are specified in the javadoc of the specific method `setDefaultXSize(long,long)` where `X` is `Boolean`, `Integer` etc. Is optional, the default value is the value specified with the `memSize` attribute if declared, else the Java size of the primitive type. Allows a cross-platform syntax using macros.

If `long` the `address` attribute can be used too.

16.1.19 Long type as cross-platform

The `long` type can be declared cross-platform with the attribute:

- `address`: if `true` declares the `long` as cross-platform, the memory size (and alignment) is the platform's address size (4 bytes in a 32 bits or 8 bytes in a 64 bits platform). Is optional, default value is `false`.

16.1.20 Primitive wrapper types

Native instances of types like `NativeBoolean`, `NativeInteger` etc contains a primitive value, a reference can be seen as a primitive pointer (`int*`, `byte*` etc). The default declaration of the contained primitive can be modified using the optional child element `<component>`:

- `<component>`: is used to redefine the default declaration of the contained primitive type (see Primitive types). Is optional, if not declared the primitive declaration is the default.

  Example: a field pointing to a char with 1 byte size

```
private NativeCharacter ptrChar = …;
```

```
<field name="ptrChar" >
    <component memSize="1" />
</field>
```

16.1.21 Primitive object types

Native instances of types like `Boolean`, `NativeBooleanObject`, `Integer`, `NativeIntegerObject` etc contains a primitive value, a reference can be seen as a primitive pointer (`int*`, `byte*` etc). The default declaration of the contained primitive can be modified using the optional child element `<component>`:

- `<component>`: is used to redefine the default declaration of the contained primitive type (see Primitive types). Is optional, if not declared the primitive declaration is the default.

  Example: a field pointing to a char with 1 byte size

```
private Character ptrChar = …;
```

```
<field name="ptrChar" >
    <component memSize="1" />
</field>
```

16.1.22 Array native variable type

Custom XML elements applied to array types (Java arrays and `NativeArray` derived, predefined or user defined):

Attribute:

- `length`: specifies the fixed length of the array, if -1 is undefined. Is optional, default value is -1 (undefined).

Element:

- `<component>`: specifies the native variable type of the array component. The native variable type can be declared using the attributes and child nodes defined in "Native variable type attributes and nodes". Is optional, if not defined the native variable type by default of the component is used. In multidimensional arrays the `<component>` node can have new `<component>` childs.

  Example: a 3 dimension array field by value with lengths 5, 10, 15

```
private int[][][] array = new int[5][10][15];
```

```
<field name="array" length="5" varConv="byValue">
    <component length="10" >
        <component length="15" />
    </component>
</field>
```

16.1.23 Pointer native variable type

Custom XML elements applied to `NativePointer` based types (predefined or user defined):

Element:

- `<pointer>`: specifies the native variable type of the pointer contained. The native variable type can be declared using the attributes and child nodes defined in "Native variable type attributes and nodes". Is optional, if not defined the native variable type by default of the pointer contained is used.

16.1.24 Behavior based native variable types

Behavior based native variable types like `java.lang.reflect.Constructor`, `java.lang.reflect.Method`, `java.lang.reflect.Field` (field-methods), and `NativeBehavior` derived classes (predefined and user defined) need a native behavior signature (how the method is seen from native side).

Attributes: same as defined in "Common native behavior attributes"

Constructor child nodes: a sequence of `<param>` nodes defining the constructor parameters. See "Element: param".

Method child nodes: a sequence of `<return>` and `<params>` defining the return and method parameters. See "Element: return" and "Element: params".

Field-method child node: the element `<fieldType>` defining the return a "value" parameter type of the field-method. See "Element: fieldType".

16.2XML JAVA CODE GENERATION DESCRIPTOR REFERENCE

The XML Java code generation descriptor structure is specified in the XMLSchema archive:

`JNIEasy.jgen.xsd`

A header example:

```
<?xml version="1.0" encoding="UTF-8"?>
<jniEasyJavaCodeGen version="1.1"
    xmlns="http://www.innowhere.com/jnieasy/jgen"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.innowhere.com/jnieasy/jgen
        ../../../schemas/JNIEasy.jgen.xsd">

    <fileGen name="MyCPPClassOnDLLGen">
...
```

16.2.1 Element: jniEasyJavaCodeGen

The root element.

- `version` attribute : is mandatory, current version is 1.1.

16.2.2 Element: include

Used to include other XML files. Useful to generate classes from command line in "batch" mode. Identical to the enhancer element.

16.2.3 Element: fileGen

Declares the file to be generated.

Attributes:

- `name`: the file name. The .java extension will be added to this name. Is required.

Contains a `<package>` child node.

16.2.4 Element: package

Declares the package of the contained classes (usually one class).

Attributes:

- `name`: the format is the same as the Java `package` declaration. Is required.

Child nodes: `<imports>` and `<class>`.

16.2.5 Element: imports

Parent element of `<import>` declarations. Identical to the enhancer element.

16.2.6 Element: import

Identical to the enhancer element.

16.2.7 Element: class

Declares the class to be generated.

Attributes:

- `name`: declares the "simple" name of the class. The format is the same as the name of a Java `class` declaration. Is required.
- `libraryPath`: specifies the name or path of the Dynamic Link Library to be used by the generated class methods.
- `extends`: specifies the class to inherit. The format is the same as the name of a Java `extends` declaration. Is optional.
- `implements`: specifies the interfaces to be implemented. The format is the same as the Java `implements` declaration. Is optional.

Child nodes: `<freeCode>`, `<field>`, `<constructor>`, `<method>`, `<fieldMethod>`.

16.2.8 Element: freeCode

Optional element to add custom code to the generated class.

Example:

```
<freeCode><![CDATA[
  public static int[] array = new int[10];
  static
  {
    for(int i = 0; i < array.length; i++) array[i] = i;
  }
]]></freeCode>
```

16.2.9 Element: field

Declares a field to be generated.

Attributes:

- `name`: the name of the field to be native. Is required.
- `class`: specifies the Java class of the field type. Is required.

16.2.10 Element: constructor

Declares a constructor to be generated.

Attributes:

- `nativeName`: specifies the exported name in the DLL to link the constructor. Is required.
- Common native behavior attributes: seen in "Common native behavior attributes".

Child nodes: `<param>` nodes specify the constructor parameters and their native variable type. Identical to enhancer (with exception of parameter names, see later).

16.2.11 Element: method

Declares a method to be generated.

Attributes:

- `name`: the name of the method. Is required.
- `nativeName`: idem `<constructor>`, but in this case is optional, if not defined the native name used is the method name.
- `methodType`: specify if the method is static (C) or instance (C++).Valid values: `C` and `CPP`. Is required.
- Common native behavior attributes: seen in "Common native behavior attributes".

Child nodes: `<return>` and `<params>` nodes specify the concrete method and native representation of return type and parameters. Identical to enhancer (with exception of parameter names, see later).

16.2.12 Element: fieldMethod

Declares a field-method to be generated.

Attributes:

- `name`: the name of the method. Is required.
- `nativeName`: idem `<method>`, but in this case the native name is the exported name of the field in the DLL, if not defined the native name used is the method name.
- Common native behavior attributes: seen in "Common native behavior attributes".

Child node: `<fieldType>` node specify the native declaration of the return and "value" parameter of the field-method. Identical to enhancer.

16.2.13 Element: param

The main difference with the enhancer version is the `name` attribute.

- `name`: specifies the name of the parameter. Is required.

Other attributes and child nodes are the same as the enhancer version, see Element: param.