Object-Oriented JNI Add-In for Microsoft Visual Studio v.7.1

1 Introduction

1.1 What is OOJNI

Object-Oriented JNI Add-In is a tool developed for MS Visual Studio. It generates Object-Oriented JNI code (OOJNI classes) for java byte code selected and includes it into the active project. Depending on Project type, Add-In generates OOJNI code in C++, managed C++, C#, J# or VB. At the same time OOJNI Add-In makes all project setting for using, compiling and running the code generated. Each OOJNI class wraps low-level JNI code to accesses to Java Class fields and methods. Data type conversions (from java-to-JNI and JNI-to-java) are hidden from Developer. JNI wrappers generated with Add-In are two types:

- wrappers generated for Java classes selected, which allow access to Java class members;
- simple wrappers which hold Java native reference. They are generated for all java types appeared in Java classes wrapped in JNI code. Simple wrapper makes a shallow copy of the java object reference in different OOJNI classes and used in Java class member calls. This approach makes possible to reduce a number of java references used in JNI code.

OOJNI package includes two runtime libraries used with JNI wrappers generated:

- **OOJNIRTVC71.dll** in C++,
- **oojni.net.dll** for .NET languages (this module is redistributed with the special Setup package).

1.2 Copyright Statement

All rights reserved © 2005-2006 Javain Ltd (Vitaly Shelest)

2 How OOJNI Works

2.1 Byte code parsing

OOJNI Add-In loads and parses raw byte code for generating JNI wrappers. It is independents of JVM installed and does not use of external modules. The parser implemented in the add-in is designed with ANTLR Tool (see http://www.antlr.org/). OOJNI code generation has three phases

- Parsing Byte Code and building a syntax tree of it
- Analyzing the syntax tree built and, if needed, modification of it
- Generating proper code (C++, managed C++, C#, J#, VB) with data collected in the syntax tree.

2.2 OOJNI Class model

OOJNI consists of OOJNI Generator that generates OOJNI wrappers for java byte code selected and a library of class helpers for data type mapping (Java-to-JNI and JNI-to-Java). These are class templates for java arrays, java string class, JVM API wrappers, etc. By default OOJNI Add-In generates JNI classes that can include wrappers for the all java class members (including inherited from base classes). However, if you want reduce the size of code generated you can restrict a list of class members to be generated. This option makes JNI code created very compact.

3 Setup OOJNI Add-In

3.1 System requirements

Before installing close all instances of MS Visual Studio running and be sure your computer meets the system requirements:

- Operating Systems
  - Windows NT 4.0
  - Windows 2000
  - Windows XP
  - Windows Server 2003
- Java Runtime Environment
  - SUN JSE1.3.x and later
  - IBM JDK1.3.x and later
- Integrated Development Environment
  - Microsoft Visual Studio v7.1

3.2 Activating OOJNI Add-In in MS Visual Studio

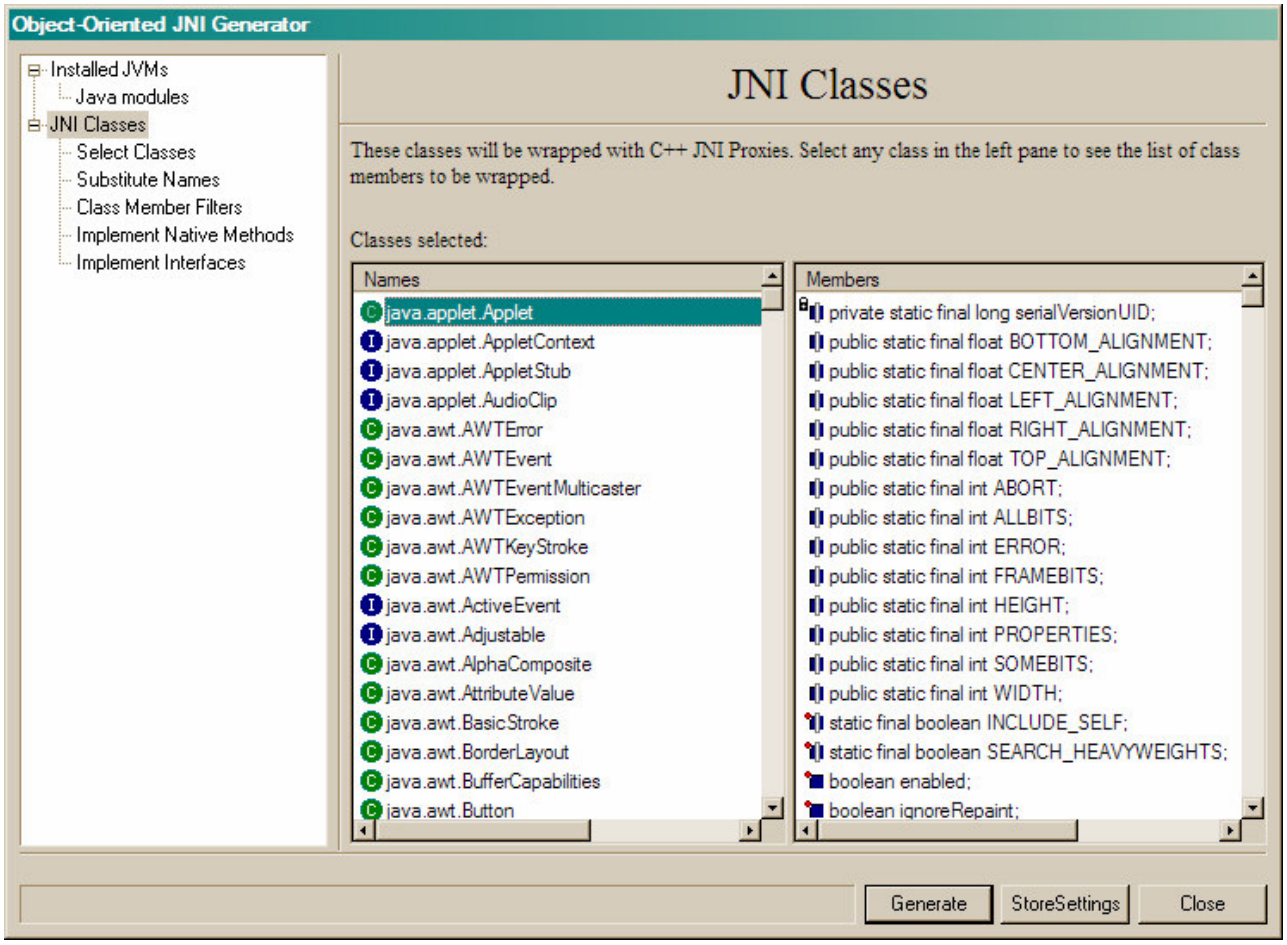
To start OOJNI Generator Dialog open click ***Tools*** and select ***OOJNI Generator***. However, the dialog will be blocked if there is no suitable active project selected. So open or create the project(s) before running OOJNI Add-In. OOJNI Add-In identifies project type selected and will generate code in the programming language that corresponds to the project type (C++, MCPP, C#, J#, VB).

4 Guide to the graphical user interface

With graphical user interface, Programmer can select java byte code for which he wants to have the JNI wrappers. A number of wrappers will be the same as a number of java classes selected. Define settings to make code compact. JNI wrapper class names are included to the project. OOJNI Add-In. makes all specific project and JVM settings.

4.1 OOJNI GUI structure

OOJNI Add-In Interface has one main dialog with a number of embedded child dialogs and the main menu as a tree for switching these dialogs (see Picture.1).



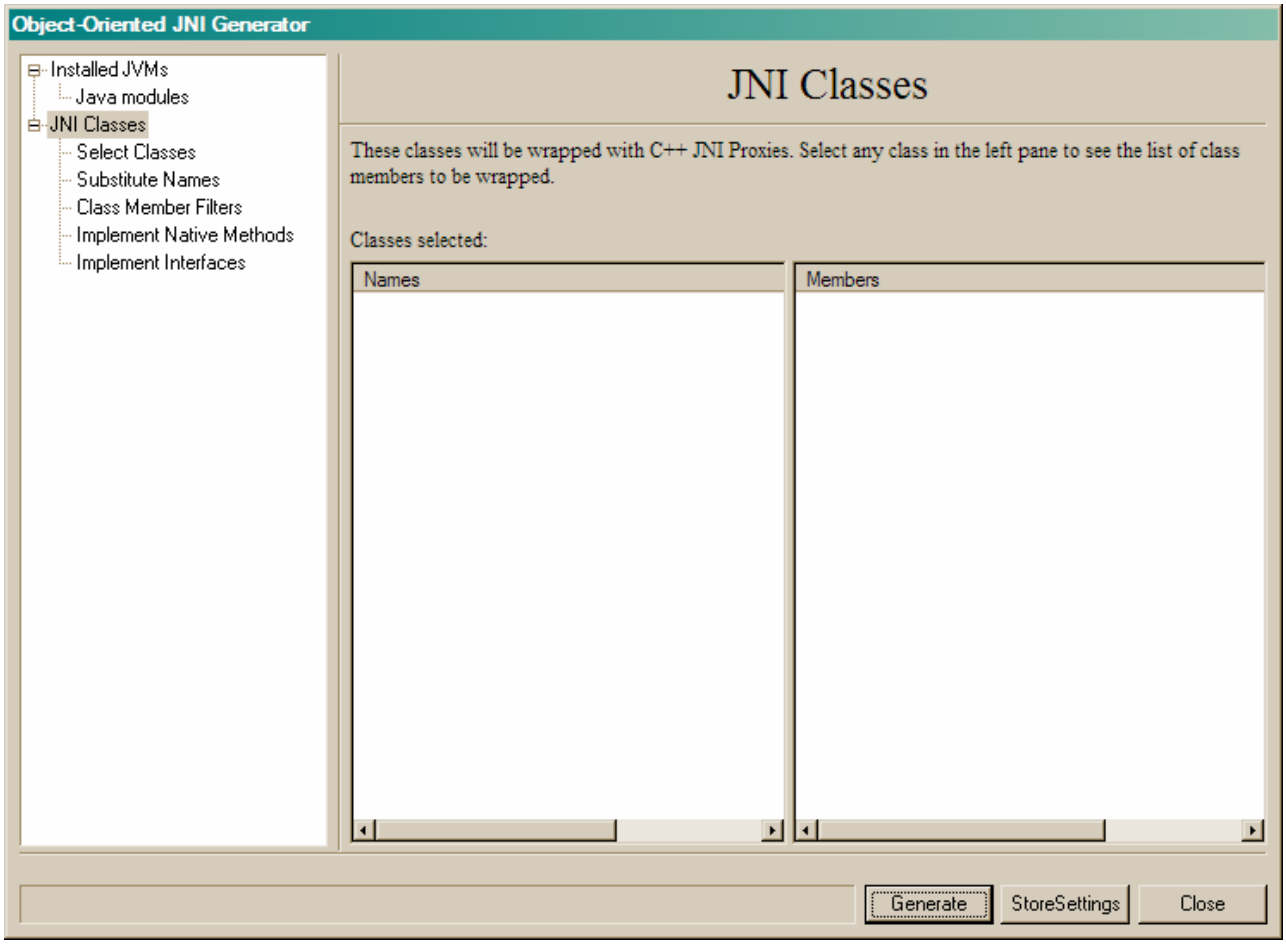
Pic.1. Main OOJNI Add-In Dialog View.

Two lists in the right are java classes selected for generating wrappers and members of the class selected. Each class name has a bitmap “C” – Class or “I”- Interface. There are three buttons always exposed:

- ***Generate*** – starts the process of generating wrappers.
- ***StoreSettings*** – stores the current OOJNI project.
- ***Close*** – closes the dialog.

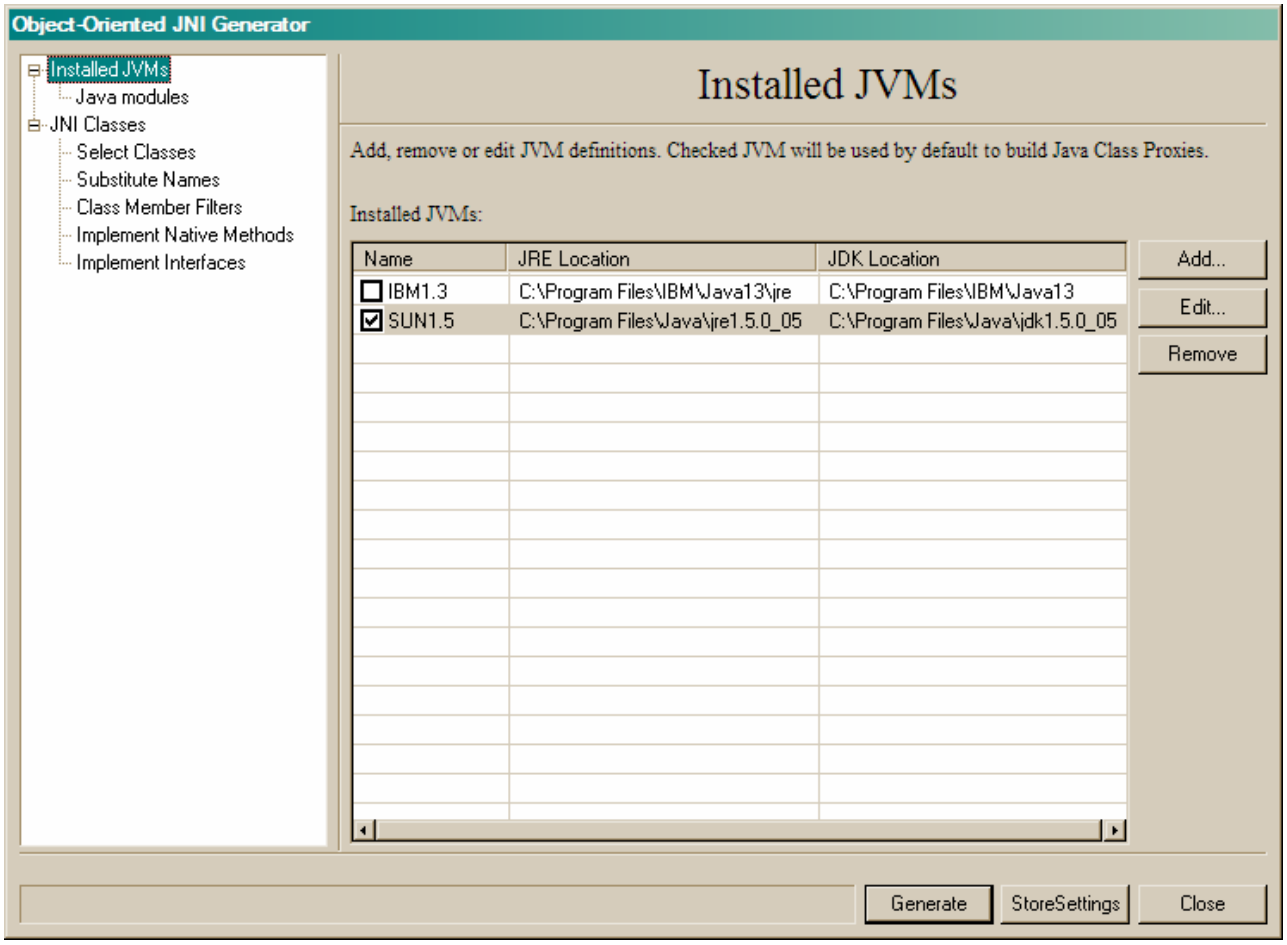
4.2 Creating OOJNI Project

Start OOJNI Dialog for a new project (see Picture.2)



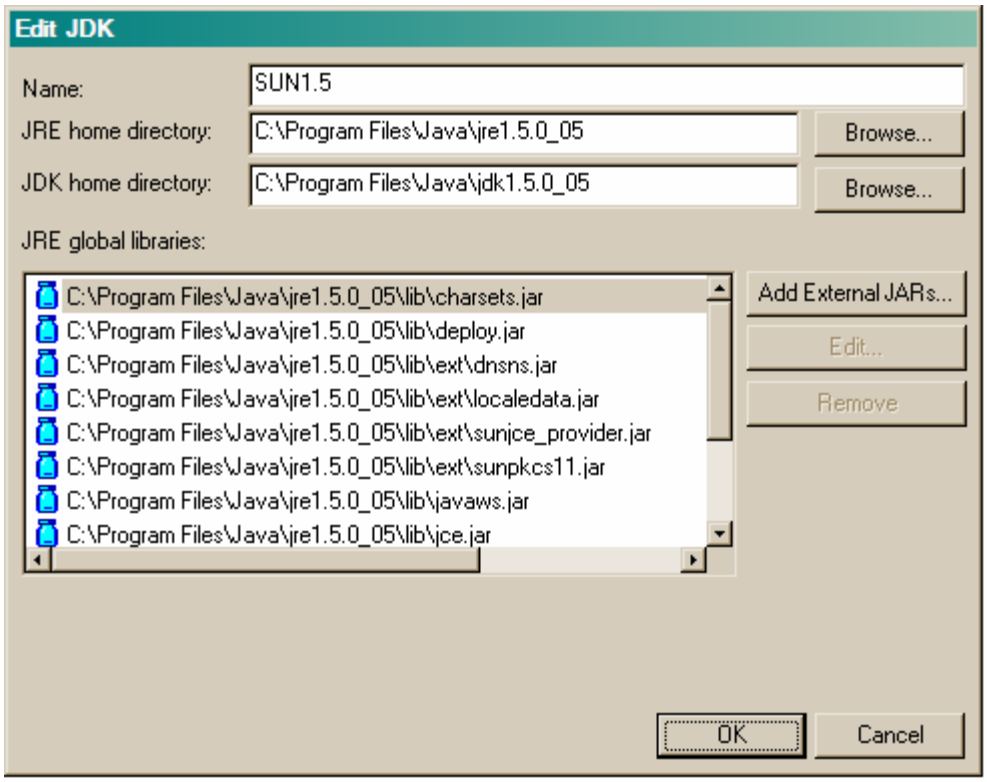
Picture.2. OOJNI Dialog for a new Project.

First, you should select JVM (if in your computer there are more than one installed). In the left tree select ***Installed JVMs*** and this view check JVM needed (see Picture.3).



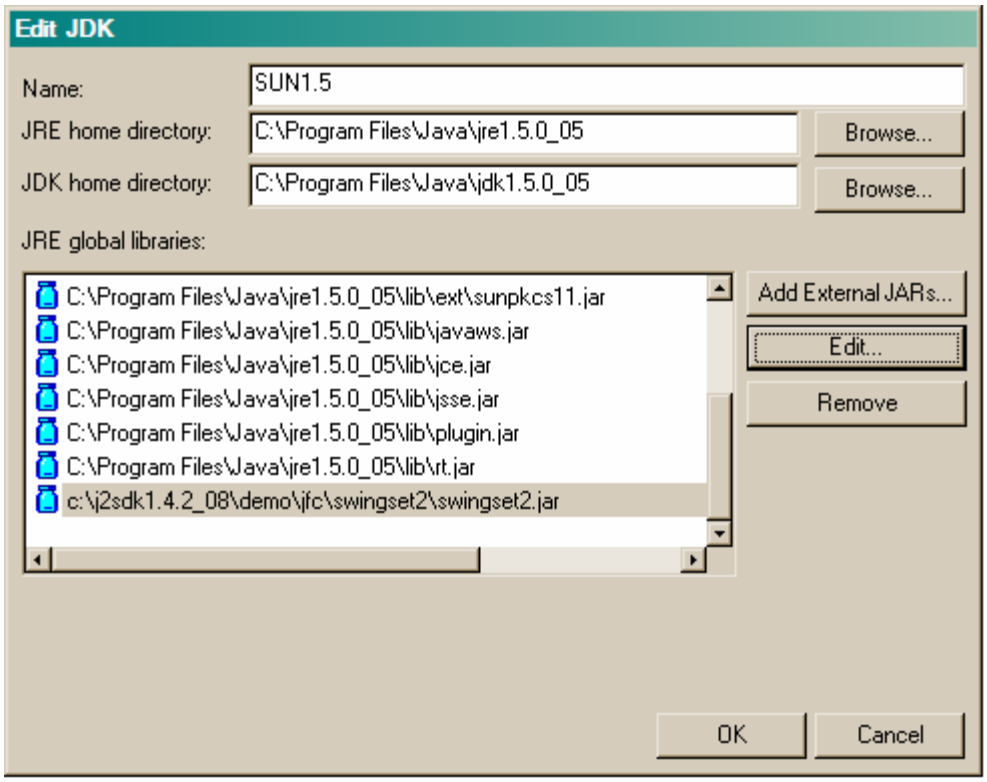
Pic.3. JVM List View

This view shows only default JVMs installed. However, if you want to select JVM version that is not shown in the list press ***Add…*** (To add a new JVM version available in your computer) or ***Edit...*** (To substitute JVM selected in the list). Let press ***Edit…***, this activates another dialog ***Edit JDK*** (Pic.4)



Pic.4. Edit JDK Path.

Correct JRE or/and JDK paths, change JDK Name in you want. Beneath there is a list of the system java modules with three buttons: ***Add External JARs…***, ***Edit…***, ***Remove***. The buttons ***Edit…*** and ***Remove*** disabled if you select a system module (Any module in this list is accessible for all OOJNI projects). To make any java mode global for all OOJNI projects add it to this list by pressing ***Add External JARs*** Button (this feature is not supported in Demo version). In the file dialog select any java module you want to be global (jar, zip, class). Let add swinset2.jar (Pic.5)



Picture.5. Adding **swingset2.jar** to the global modules list.

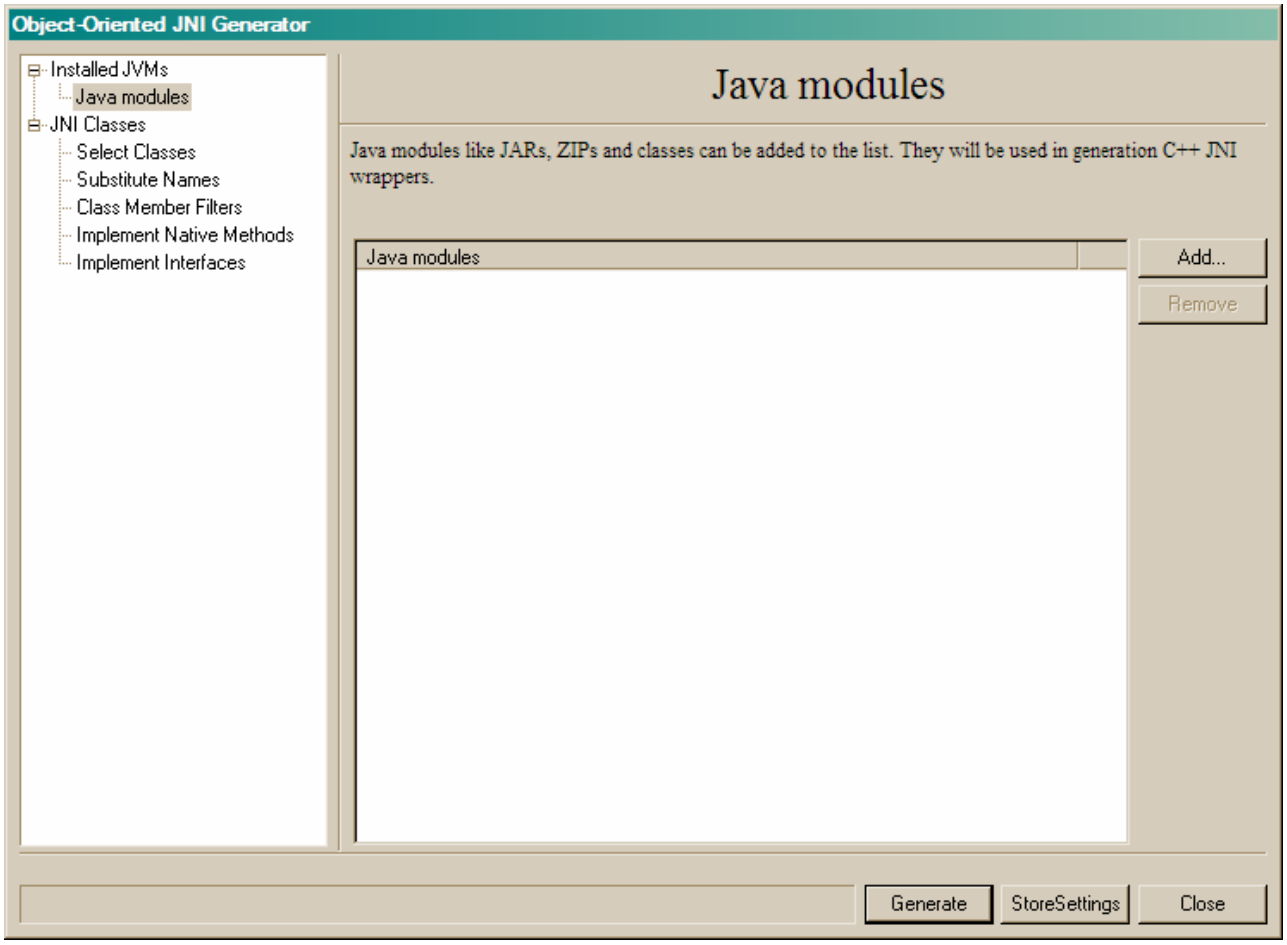
Here ***Edit…*** and ***Remove*** buttons enabled and this item you can remove or substitute.

4.3 Connecting to JVM installed

In the JVM List View, (see Picture.3) the checked JVM is used for generating JNI wrappers. It means that classes from system/global modules OOJNI generator looks for without special setting. If you set check of another JVM then OOJNI Add-In reloads all classes selected in the list of view in the Picture.1.

4.4 Loading java modules

Now select ***Java Modules*** item in the left Tree to activate the module selection view (Pic.6).

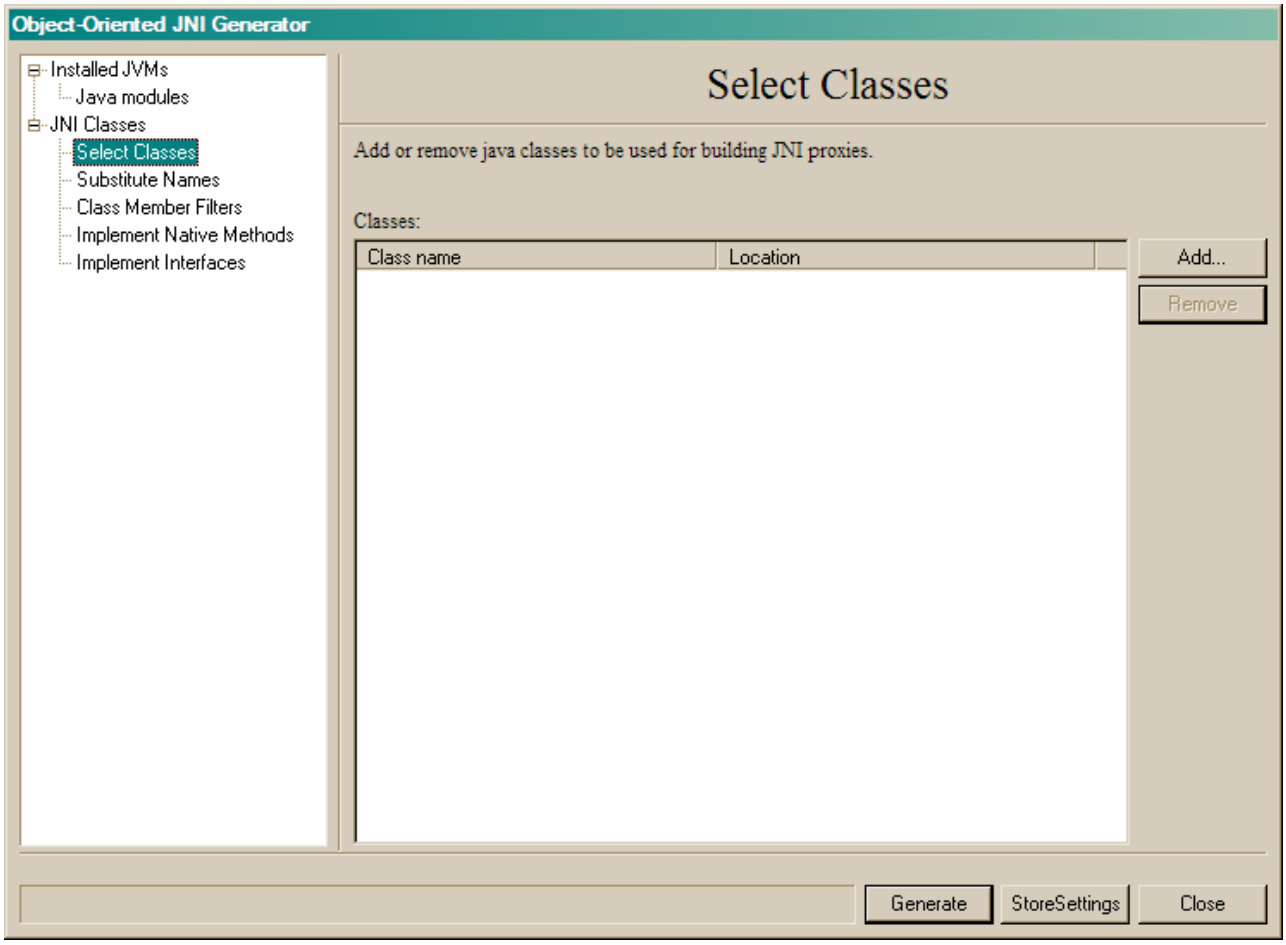


Picture.6. Java Modules List View

With buttons, ***Add…*** and ***Remove,*** create a list of java modules (jar, zip, class) you want to use in OOJNI Project.

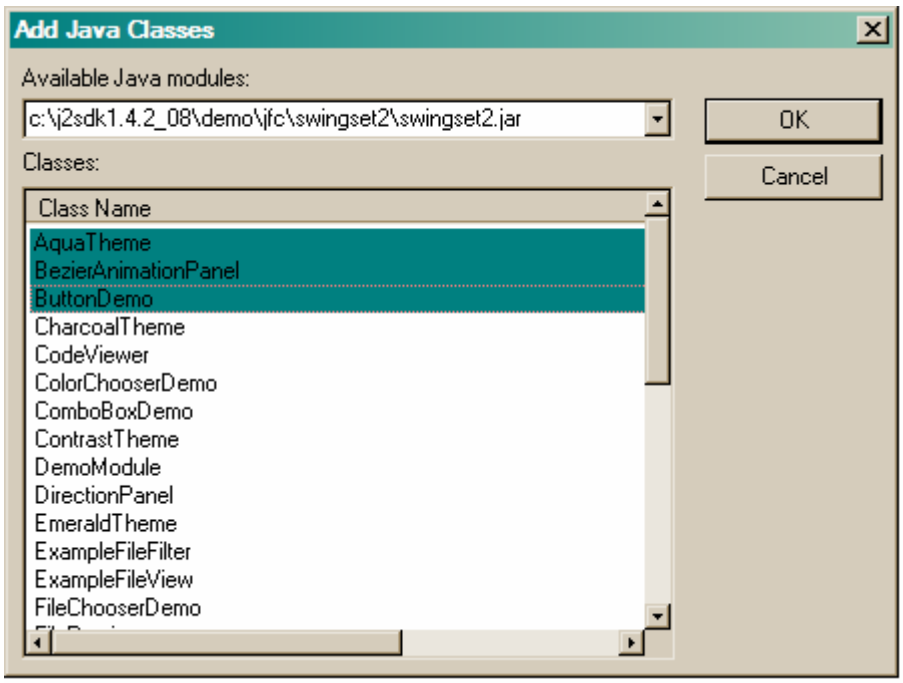
4.5 Selecting java classes to be wrapped with JNI code

Having done the previous steps, you are ready for selecting byte code for JNI wrappers. In the left tree view click on ***Select Classes*** item and the ***Select Classes*** View will appear (Pic.7).



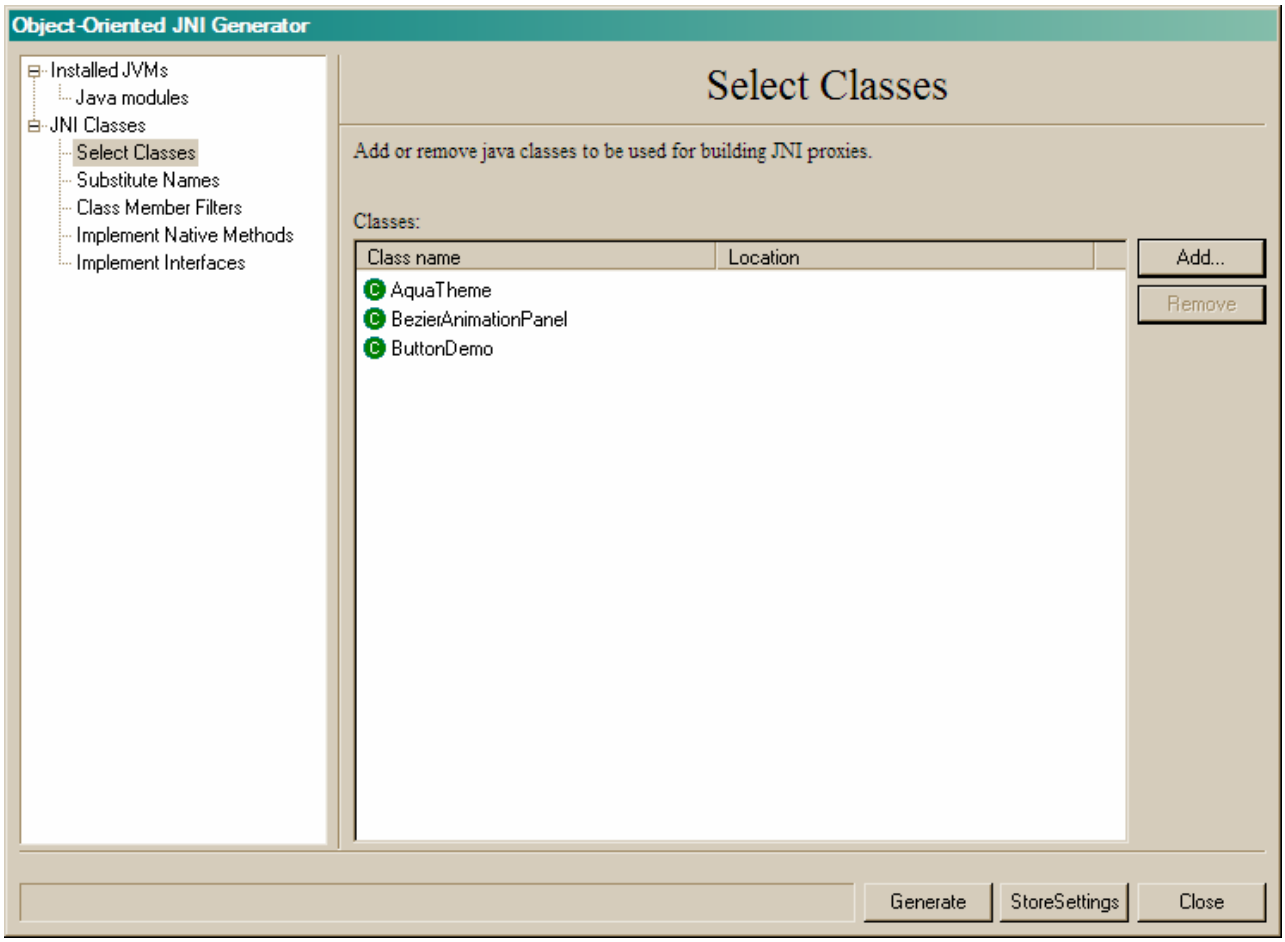
Picture.7. ***Select Classes*** View

To add byte code to be wrapped open with ***Add…*** button ***Add Java Classes*** Dialog (see Pic.8).



Picture.8. ***Add Java Classes*** Dialog

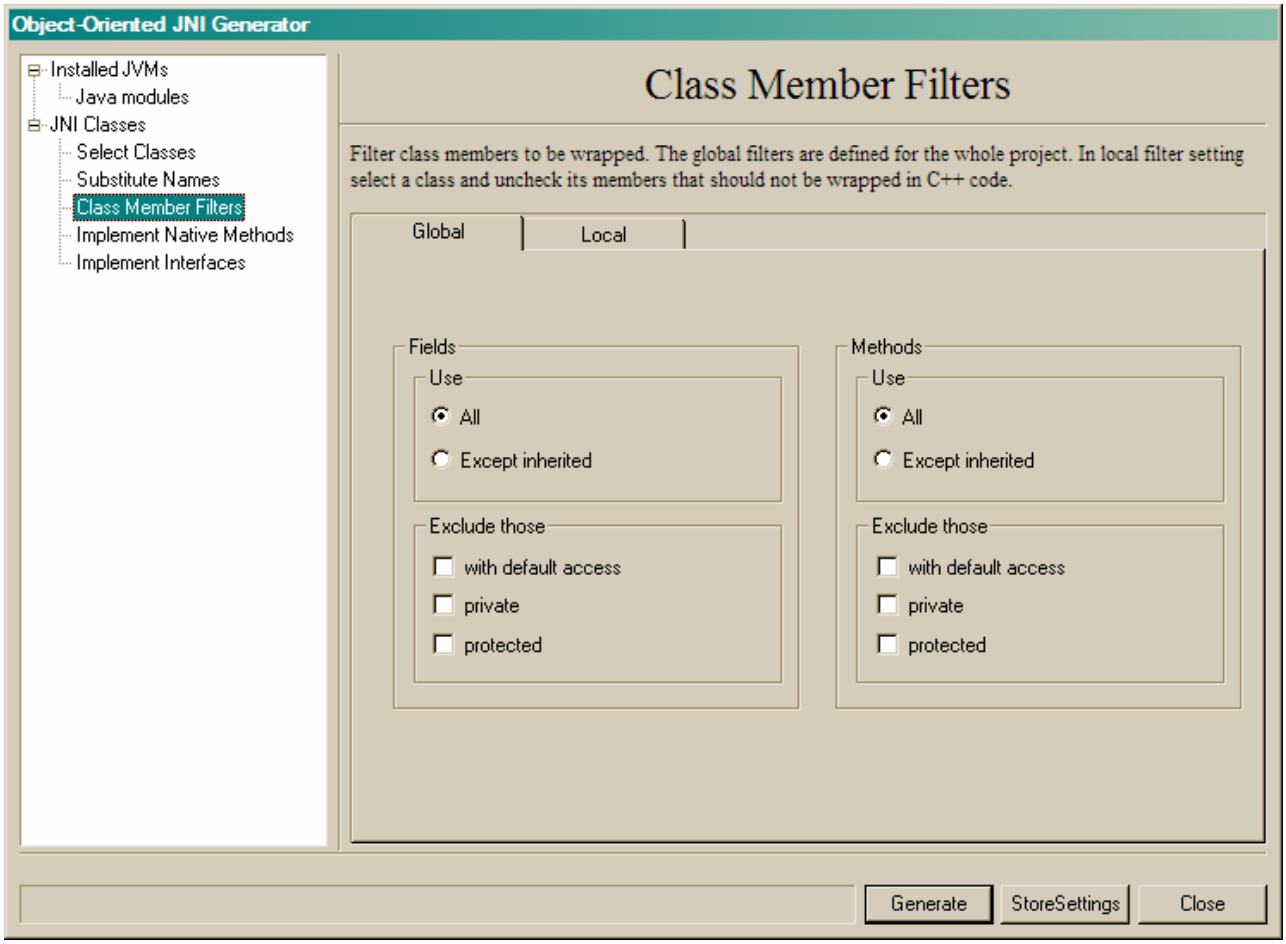
Select jar/zip module from the combo box and highlight class names you want to use. Then click ***OK*** button (see Picture.9).



Picture.9. Byte code selected for generating JNI wrappers.

4.6 Filtering class members while code generating

By default OOJNI Add-In creates wrappers for all class members (including inherited). However, the most of this code may be useless in the current project. To exclude this overhead from the process of generating JNI wrappers you should set member filters. To do this open ***Class Member Filters*** View (see Picture.10)



Pic.10. ***Class Member Filters*** View (Global Filters)

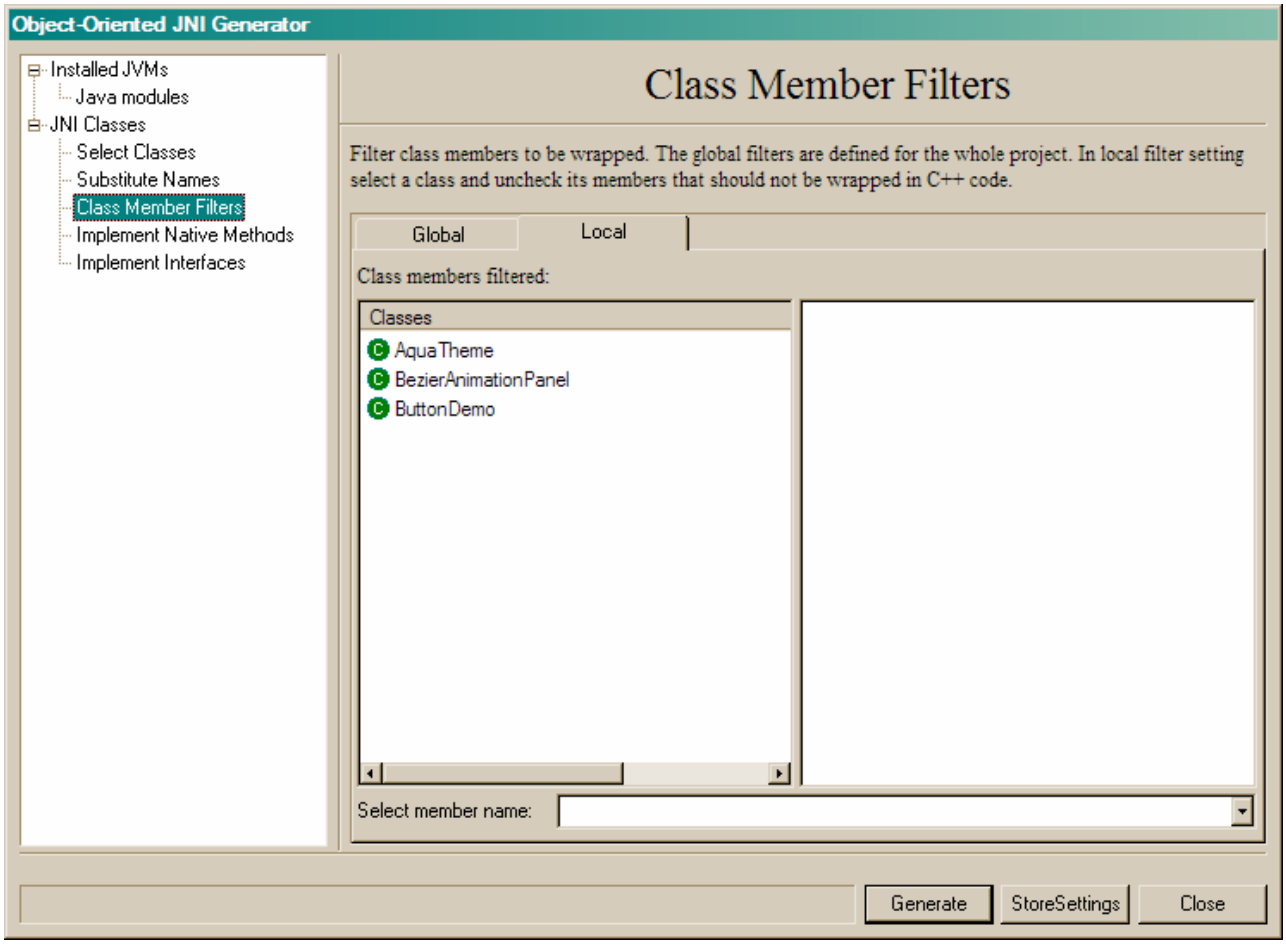
There are two types of filters

- Global filters that define filter conditions for the whole project.
- Local filters can be set for specific classes.

Global filters are defined for fields and methods separately. You have two filtering options:

- Usage of all members (***All*** radio button) or exclude members inherited from base classes (***Except inherited*** radio button),
- Exclude members by access type (default access, private, protected).

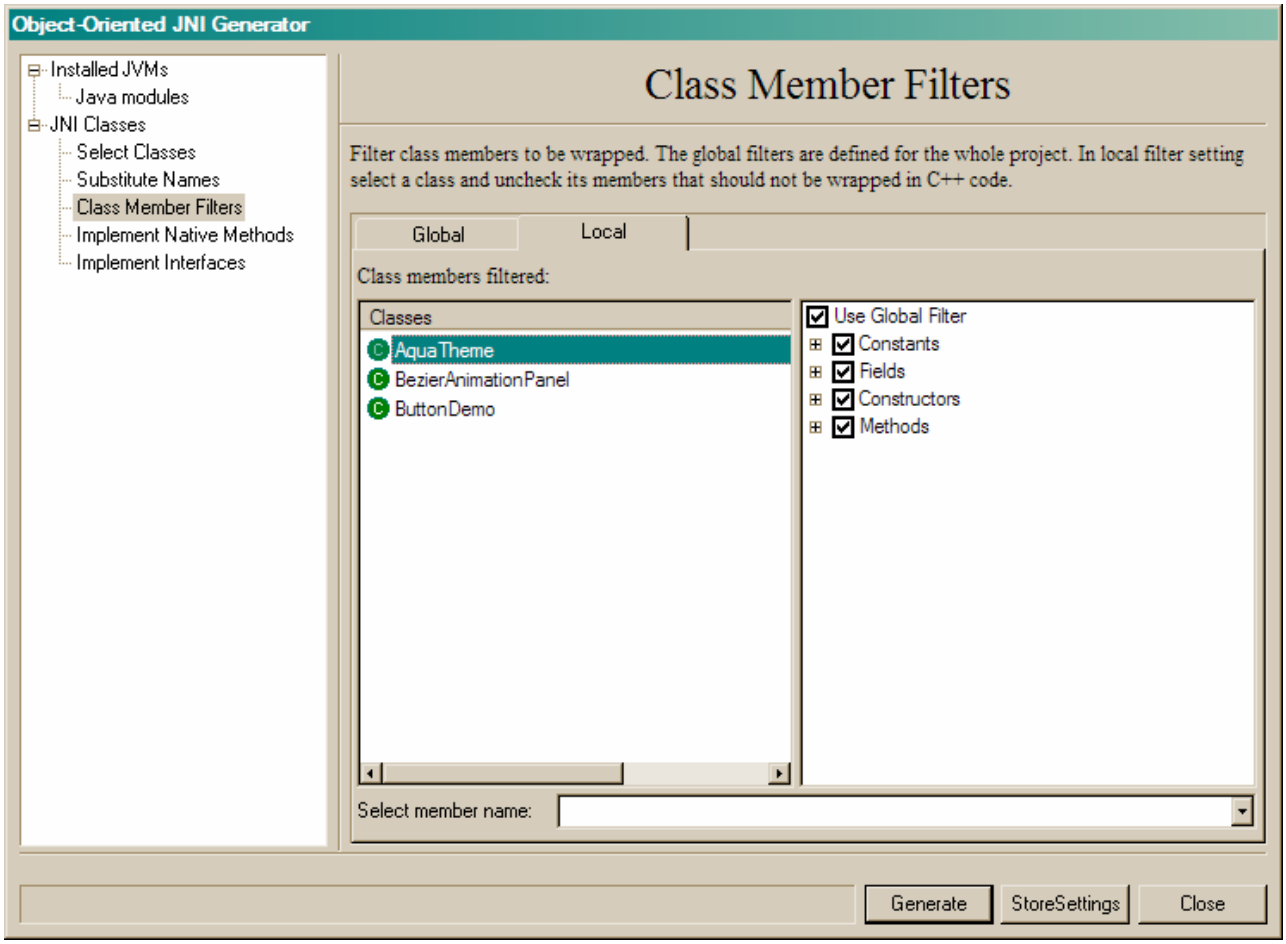
Now click on ***Local*** Tab to the view local filters settings (see Picture.11). Here select specific class name and set member filers you want (see Picture.12).



Picture.11. ***Local Filter*** view.

In the right TreeView of the panel (see Picture.12) will appear a list of members grouped by

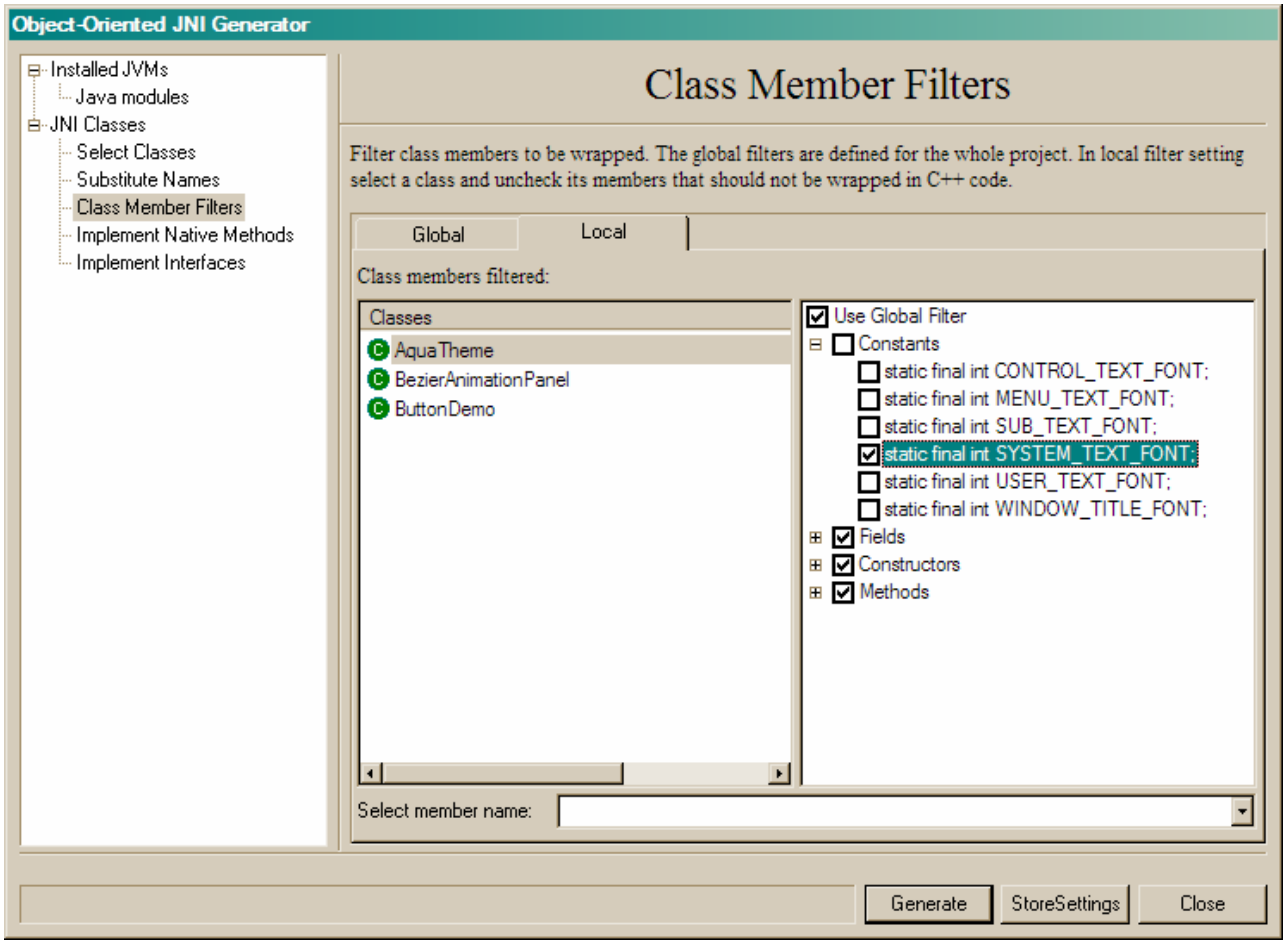
- Constants, these members are represented in code as constants and accessed without JNI code use,
- Fields
- Constructors
- Methods



Picture.12. Specific Class Members Filter View

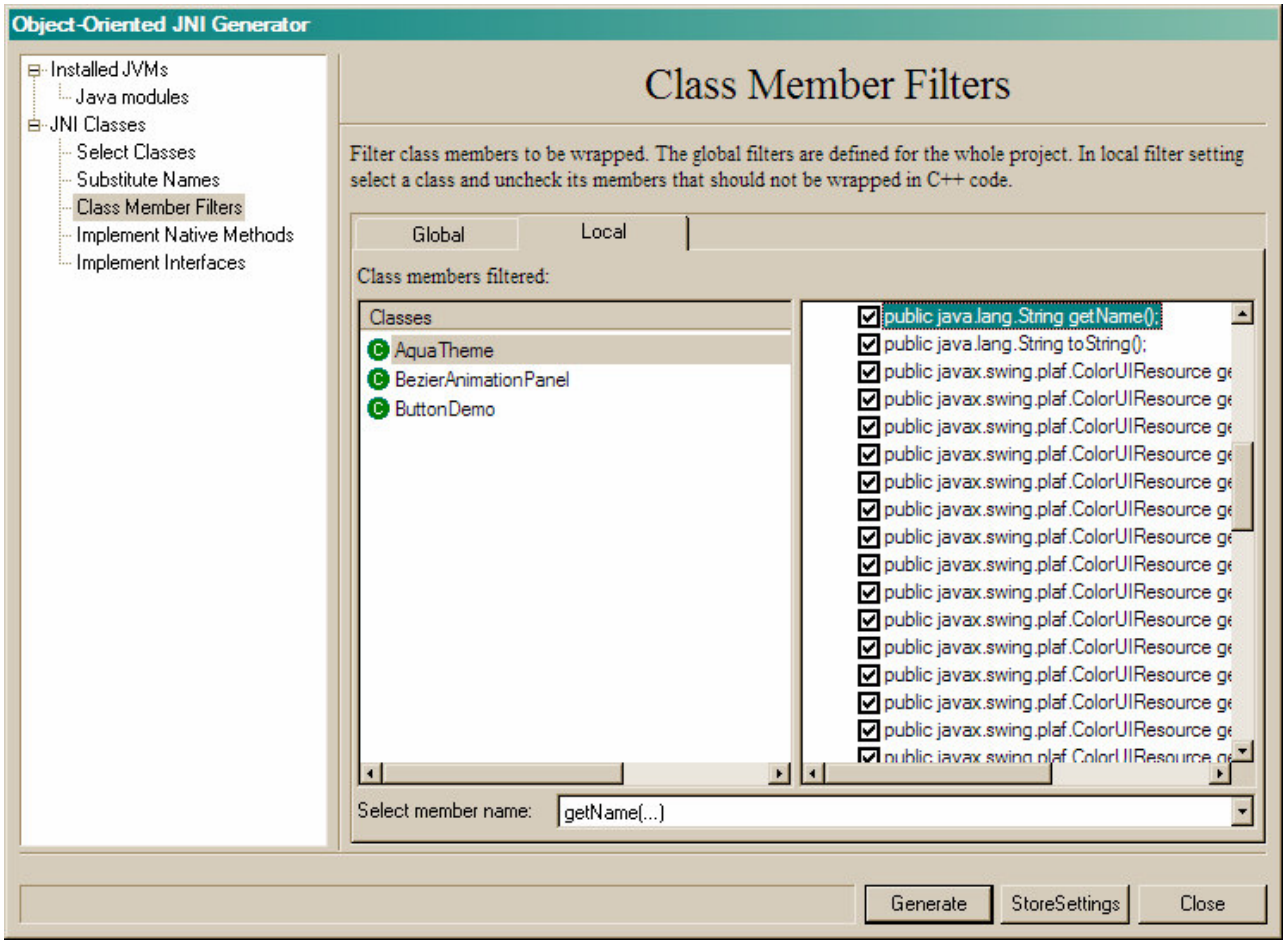
Any group checked means that all members of it are wrapped. If ***Use Global Filter*** item is checked then this tree view includes only members filtered with Global Filters. To do selection of specific members for wrapping, (see Picture.13)

- Uncheck the group,
- Expand it
- Check only members needed.



Pic.13. Filtering specific members

If the class you selected has a huge number of members, then finding specific member name in the tree view takes much time. In this case, you can use Search Helper beneath the view (see Picture.14). Select a member name you want in this combo box to highlight the first class member with this name in the tree view.



Pic.14. Class Member Name Highlighting

4.7 Disambiguing code generated

In some cases java classes have features that not supported in destination language (C++, MCPP, C#, J#, VB). Let see the java class:

```
public class A {
public int sfm;
public long sfm;
public void sfm(){};
}
```

Here are the same name is used for the method and fields. There are two fields with the same name but with different type. For example, such declarations are not acceptable in C++ code. So while generating an OOJNI class in C++ Add-In makes the code:

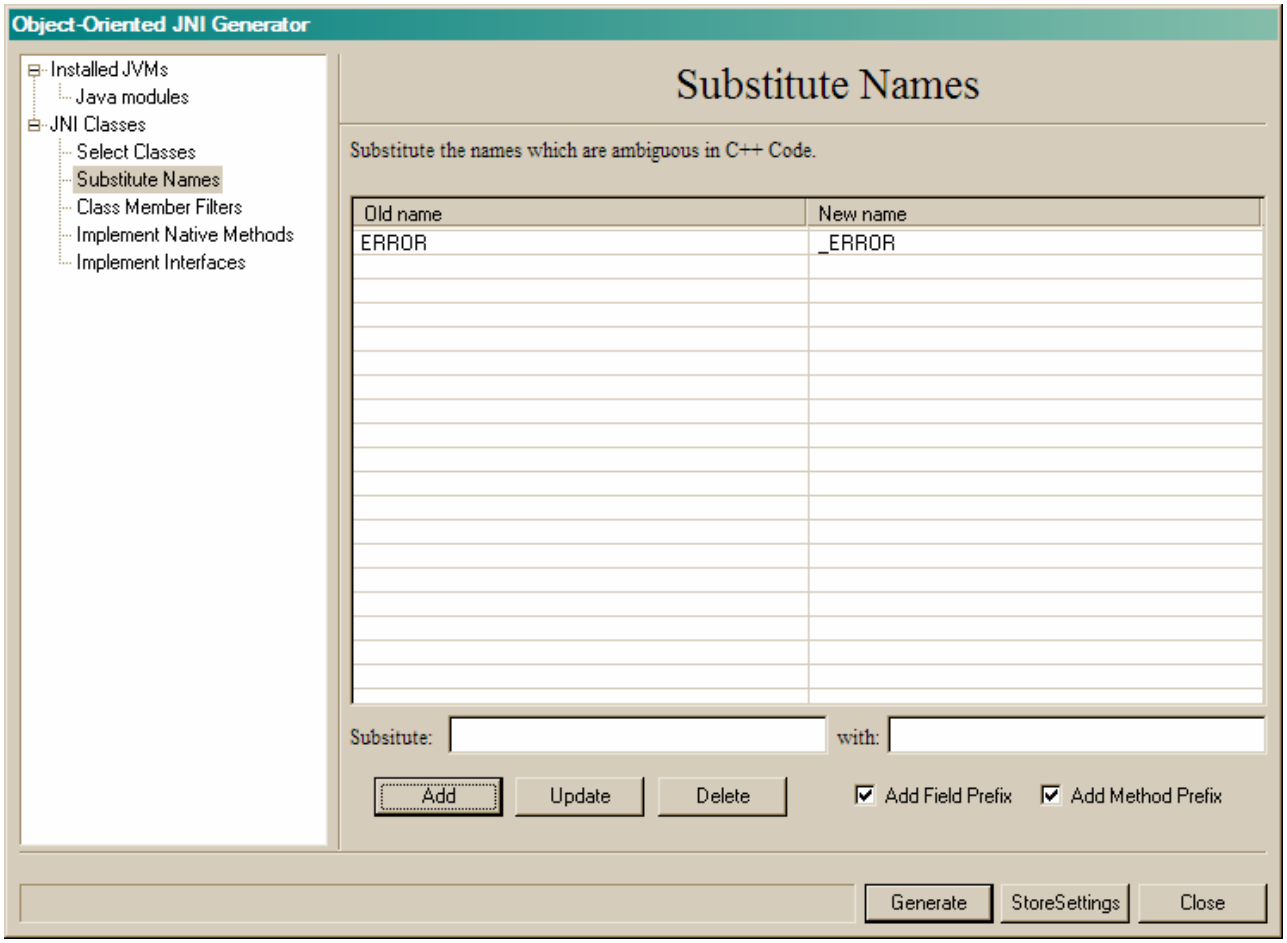
```
class A {
public:
    . . .
    jint sfm;
    . . .
```

```
    jlong sfm1;
    . . .
}
```

Alternatively, you can select options to prefix field and method names like:

```
class A {
public:
    . . .
    jint fld_sfm;
    . . .
    jlong fld_sfm1;
    . . .
    void mth_sfm(){};
public void sfm(){};
}
```

Another problem is that C++ preprocessor may have the same names macros as ID/key names in java code. For example, in C++ code the name ERROR used by the C++ preprocessor and the class generated with OOJNI Add-In may have field name ERROR. To solve this problem create substitution name list. For example, substitute the name ERROR with _ERROR (see Picture.16). To prefix field and/or method names set check boxed in this view.

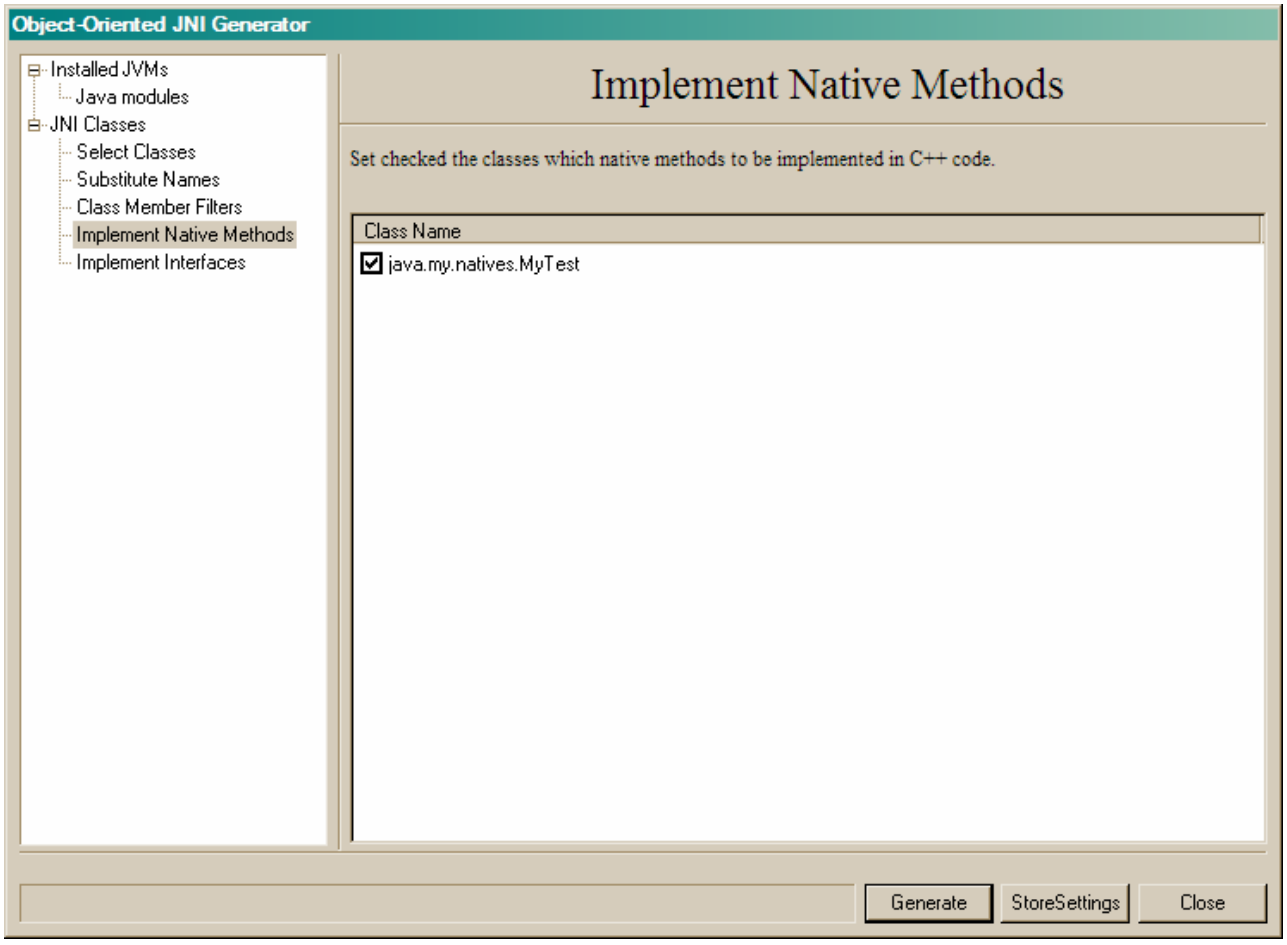


Pic.15. ***Substitute Names*** View

4.8 Implementing Java Native Methods

Java is a popular cross-platform programming language in use today, with ports available for almost any operating system. However, with the panacea of platform-independence comes the price of potentially reduced performance and the inability to direct access hardware-dependent features, a major problem in the field of 3D graphics and other hardware-accelerated areas.
To address these issues and provide the developer with a way to access this important functionality, Java provides a mechanism called *native methods*, an technique that allows your code to seamlessly switch execution from Java into compiled, platform-specific code. OOJNI API provides you with means to implement native methods in C++, MCPP, C#, J#, VB in Object-Oriented manner. Therefore, you can use java class wrappers in programming your native methods that makes your code simple for reading and debugging.

To implement this class native methods in your C++, MCPP, C#, J# or VB code insert it with ***Select classes*** Dialog to the list of classes selected, open View ***Implement Native Methods*** and set this class checked. (See Picture 16).



Pic.16. ***Implement Native Methods*** View

OOJNI Add-In includes Java classes with native methods into ***Implement Native Methods*** View. Check classes you want to generate JNI implementation of native methods and press ***Generate*** button. OOJNI Add-in

- Generates new OOJNI wrapper's H and CPP files for C++ or CS, JSL, VB files;
- Adds OOJNI wrapper file names to the project selected;
- Inserts all project settings needed.

Let, for example, implement native methods in `java.my.natives.MyTest`

```
package java.my.natives;

public class MyTest {
      public native void impl_k_ev(Object o);
      public native int[] impl_ev(Integer a0,
                                  double a1, Object a2);
      public native byte impl_ev(Long a0, Float a1);
      public native void impl_ev(Long a0, Float a1,
                                 Object[][] a);
      public static native String[] StaticMth(Class[] cc);
}
```

4.8.1 Implementing native methods in .NET languages (C#, J#, VB, MCPP)

When you generate with OOJNI Tool implementation of Java native methods in a .NET language each Java class with native methods wrapped with three class proxies:

1) <*Java Class Name*>**Proxy**. <*language extension*>
2) <*Java Class Name*> **Natives**.<*language extension*>
3) <*Java Class Name*>**NImpl**. <*language extension*>

The class (1) is regular Java class wrapper, which is generated for any bytecode selected (but native method wrappers are not implemented and **abstract**). The class (2) has only static methods named with **javah.exe** style. JVM calls these methods directly. Implementation of these methods calls corresponding method in the class (3) which is written by Developer. OOJNI Tool generates only its template (see examples in the package for MCPP, C#, J#, VB).

4.8.2 Implementing native methods in C++

OOJNI Tool generates static method implementations in javah.exe style. Each of them wraps jobject reference with C++ class `CPP_Java_Bridge::java::my::natives::MyTestJNIProxy`. In this class you should only implement methods that correspond to java class native methods:

```
CPP_Java_Bridge::java::lang::StringArray1D
CPP_Java_Bridge::java::my::natives::MyTestJNIProxy::StaticMth
(
    CPP_Java_Bridge::java::lang::ClassArray1D
);
```

```
CPP_Java_Bridge::JIntArray1D
CPP_Java_Bridge::java::my::natives::MyTestJNIProxy::impl_ev
(
    CPP_Java_Bridge::java::lang::Integer,
    jdouble,
    CPP_Java_Bridge::java::lang::Object
);
jbyte CPP_Java_Bridge::java::my::natives::MyTestJNIProxy::impl_ev
(
    CPP_Java_Bridge::java::lang::Long,
    CPP_Java_Bridge::java::lang::Float
);
void CPP_Java_Bridge::java::my::natives::MyTestJNIProxy::impl_ev
(
    CPP_Java_Bridge::java::lang::Long,
    CPP_Java_Bridge::java::lang::Float,
    CPP_Java_Bridge::java::lang::ObjectArray2D
);
void CPP_Java_Bridge::java::my::natives::MyTestJNIProxy::impl_k_ev
(
    CPP_Java_Bridge::java::lang::Object
);
```

5 Programming with OOJNI in C++

See OOJNI help file.

6 C++ Examples

All OOJNI examples are Console Applications. As java code has no the main thread loop each example (except ***EmbeddedAWTFrame*** example) has a primitive loop:

```
    while(true){
        Sleep(50);
    };
```

Before the first use of any JNI wrapper class you should activate JVM (attach JVM to the current thread). So each example starts Default JVM installed on your computer:

```
JVM::load();
```

The rest example source looks like java code. Package names are mapped to C++ namespaces and fields are mapped to properties with names prefixed: “get_”, “put_”.

6.1 RowLayoutTest

This code creates Frame window with Flow Layout and two push buttons inside.
Java classes wrapped:

java.awt.BorderLayout,
java.awt.Button,
java.awt.Color,
java.awt.Frame,

java.awt.Panel

Implemented interfaces:
java.awt.event.ActionListener
java.awt.event.WindowListener

The main code:

```
#include "stdafx.h"
#include "defproxies.h"
#include <stdio.h>

using namespace CPP_Java_Bridge;
using namespace jni_helpers;

// Implement WindowListener interface
class WindowListener: public
CPP_Java_Bridge::java::awt::event::WindowListenerJNIImpl {
public:
    void windowClosing(CPP_Java_Bridge::java::awt::event::WindowEvent p0);
};
// Implement Left Button ActionListener interface
class LeftActionListener:public java::awt::event::ActionListenerJNIImpl {
public:
    void actionPerformed(java::awt::event::ActionEvent p0);
};

// Implement Right Button ActionListener interface
class RightActionListener:public java::awt::event::ActionListenerJNIImpl {
public:
    void actionPerformed(java::awt::event::ActionEvent p0);
};

void WindowListener::windowClosing(java::awt::event::WindowEvent p0)
{
    exit(0);
}

void LeftActionListener::actionPerformed(java::awt::event::ActionEvent p0)
{
    printf("Left BUTTON Clicked\n");
}

void RightActionListener::actionPerformed(java::awt::event::ActionEvent p0)
{
    printf("Right BUTTON Clicked\n");
}

class SouthPanel : public java::awt::PanelJNIProxy {
      java::awt::ButtonJNIProxy button1;
      java::awt::ButtonJNIProxy button2;
    LeftActionListener lbl;
    RightActionListener rbl;
public:
```

```
    SouthPanel() {
            button1.setLabel("Button1");
            button2.setLabel("Button2");
            add((java::awt::Component)button1);
            add((java::awt::Component)button2);
        setBackground(java::awt::ColorJNIProxy::get_red());
        button1.addActionListener(lbl);
        button2.addActionListener(rbl);
      }
};

class FlowLayoutTest : public java::awt::FrameJNIProxy {
      SouthPanel south;
    WindowListener l;
public:
    FlowLayoutTest(LPSTR title):java::awt::FrameJNIProxy(title) {
        setBackground(java::awt::ColorJNIProxy::get_green());
      add((java::awt::Component)south,
      java::awt::BorderLayoutJNIProxy::get_SOUTH());
        addWindowListener(l);
      }
};

int main(int argc, char* argv[])
{

    JVM::load();
    //
    FlowLayoutTest test("FlowLayout Test1");
    test.setBounds(0, 0, 320, 240);
    test.show();

      // Main Primitive Loop
    while(true){
        Sleep(50);
    };
      return 0;
}
```

6.2 Grid11

This is a demo of GridLayout painted.
Java classes wrapped:
java.awt.Color,
java.awt.Frame,
java.awt.GridLayout
java.awt.Label

Implemented interfaces:
java.awt.event.WindowListener

The main code:

```
#include "stdafx.h"
```

```
#include "defproxies.h"

using namespace CPP_Java_Bridge;
using namespace jni_helpers;

// Implement WindowListener interface
class WindowListener: public
CPP_Java_Bridge::java::awt::event::WindowListenerJNIImpl {
public:
    void windowClosing(CPP_Java_Bridge::java::awt::event::WindowEvent p0);
};

void WindowListener::windowClosing(java::awt::event::WindowEvent p0)
{
    exit(0);
}

class Grid11 : public java::awt::FrameJNIProxy {
    java::awt::GridLayoutJNIProxy gl;
    java::awt::LabelJNIProxy l0;
    java::awt::LabelJNIProxy l1;
    java::awt::LabelJNIProxy l2;
    java::awt::LabelJNIProxy l3;
    WindowListener l;
public:
    Grid11(LPTSTR title):java::awt::FrameJNIProxy(title),
        gl(1, 1),
        l0("1*1", java::awt::LabelJNIProxy::CENTER),
        l1("2*2", java::awt::LabelJNIProxy::CENTER),
        l2("3*3", java::awt::LabelJNIProxy::CENTER),
        l3("4*4", java::awt::LabelJNIProxy::CENTER)
    {
        setLayout(gl);
        l0.setBackground(java::awt::ColorJNIProxy::get_blue());
        l1.setBackground(java::awt::ColorJNIProxy::get_red());
        l2.setBackground(java::awt::ColorJNIProxy::get_blue());
        l3.setBackground(java::awt::ColorJNIProxy::get_red());
        add((java::awt::Component)l0);
        add((java::awt::Component)l1);
        add((java::awt::Component)l2);
        add((java::awt::Component)l3);
        addWindowListener(l);
    }
};

int main(int argc, char* argv[])
{
    JVM::load();

    Grid11 frame("Grid11 Test");
    frame.setBounds(100, 100, 200, 200);
    frame.show();

      // Main Primitive Loop
    while(true){
        Sleep(50);
    };
```

```
      return 0;
}
```

6.3 HelloWord

Shows Frame with Label component.
Java classes wrapped:
java.awt.Frame
java.awt.Label

Implemented interfaces:
java.awt.event.WindowListener

The main code:

```
#include "stdafx.h"
#include "defproxies.h"

// Implement WindowListener interface
class WindowListener: public
CPP_Java_Bridge::java::awt::event::WindowListenerJNIImpl {
public:
    void windowClosing(CPP_Java_Bridge::java::awt::event::WindowEvent p0);
};

void WindowListener::windowClosing(java::awt::event::WindowEvent p0)
{
    exit(0);
}

class Hello : public java::awt::FrameJNIProxy {
    // A label, in C++, can not call constructor at here, call it
    // at the enclosing class's ctor-initilizer.
    java::awt::LabelJNIProxy hello;
    WindowListener l;
public:
    Hello()
        : java::awt::FrameJNIProxy("Hello"),
        hello("Hello, world!", java::awt::LabelJNIProxy::CENTER) { // construct
the label
        // add the label into the frame.
        add((java::awt::Component)hello);
        addWindowListener(l);
    }
};

int main(int argc, char* argv[])
{
    JVM::load();

    Hello hello;
    hello.pack();
    hello.show();

      // Main Primitive Loop
```

```
    while(true){
        Sleep(50);
    };      return 0;
}
```

6.4 EmbeddedAWTFrame

Demonstrates embedding of AWT components into MFC Application. Here the method ***javax.swing.SwingUtilities.invokeLater(…)*** used only for illustration purposes.

Java classes wrapped:
java.awt.Buttton
java.awt.Color
java.awt.Panel
javax.swing.SwingUtilities
sun.awt.windows.WEmbeddedFrame

Implemented interfaces:
java.lang.Runnable

In the code generated with MS Visual Studio to embed java code we did

- Added loading JVM before the dialog starts: in EmbeddedAWTFrame.cpp added *jni_helpers::JVM::load();* to "InitInstance()" function
- Created Runnable class with the class generated *CPP_Java_Bridge::java::lang::RunnableJNIImpl* to implement the method *run()*
- In the method *OnShowWindow()* the java code is embedded into C++ Dialog