Lycia Development Suite

Lycia Java Interface
Version 6 – July 2011
Revision number 1.00

Querix Lycia

Java Interface

Part Number: 024-006-100-011

Elena Krivtsova / Ross Freemantle

Technical Writer / Developer

Last Updated 04 August 2011

Lycia Java Interface

Part Number: 024-006-100-011

Published by:

Querix (UK) Limited. 50 The Avenue, Southampton, Hampshire,
SO17 1XQ, UK

Publication history:

July 2011:	Beta edition
August 2011:	First Edition

Last Updated:

August 2011

Documentation written by:

Elena Krivtsova, Ross Freemantle

Notices:

Table of Contents

Introduction

Java Interface introduced in Lycia allows you to utilize your Java code in your 4GL programs. Earlier only the integration of the C code was included. However, with the development of the Information Technologies, the Java language becomes more and more popular and lately it has become the cutting edge of the development industry. Thus Querix decided to introduce the Java integration in order to bring the 4GL language on a new level and to integrate 4GL programs with most recent and useful technologies there are.

Case Sensitivity of Java Identifiers

Unlike Lycia 4GL, the Java language is case sensitive. Therefore, when you write the name of a Java package, class, method, or field in a .4gl source, it must match the exact name as if you were writing a Java program. Attempting to compile a .4gl source containing a Java identifier typed in the wrong case will result in a compilation error.

Adjusting Java Environment

Before you will be able to use Java code in your 4GL programs, you need to install the JRE and to specify the environment variables pointing to the java files you will be using and to the necessary libraries.

Requirements

You need to install Java and set the paths to the dynamic linker and Java resources.

Installing Java

To be able to import and use Java classes you need to install Java Runtime Environment (JRE) of version 6 or above. The JRE contains all the functionality necessary to use the Java interface, because Java interface utilizes the already compiled .class files or Java packages containing .class files. The creation and compilation of Java classes that can be performed by JDK as well as Java language syntax are outside of the scope of this manual.

Setting the Dynamic Linker

You need to configure your environment to let the dynamic linker find the libjvm.so (or libjvm.dylib) shared library on UNIX or the jvm.dll on Windows.

If you plan to launch your applications in the GUI mode, you need to set the same path in the inet.env file. For more information about this file see "Lycia Developers Guide".

OnLinux/Intel

You should add $JRE_HOME/lib/i386/serverpath to the **LD_LIBRARY_PATH** variable (or other directory where **libjvm.so** is located).

On Mac OSX

You should add /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Libraries to the **DYLD_LIBRARY_PATH** variable (or other directory where **libjvm.dylib** is located).

On Windows

On Windows you should add the directory where **jvm.dll** is located to the system **PATH** variable. For Java 32 bit it is %JAVA_HOME%\bin\client and for Java 64 bit it is %JAVA_HOME%\bin\server.

Note: On any Windows OS, if out use Java 6 you'll also need to add %JAVA_HOME%\bin to the system PATH.

Locating Java Resources

All the Java resources which contain the classes you intend to use in Java interface should be specified in the **CLASSPATH** environment variable. This variable should point at the directory or directories where compiled .class files or Java packages containing .class files are located.

If you want to use a custom class you should create a .java file with this class, compile it into the .class file and set the path to it in the CLASSPATH variable.

If you only work with the Java classes included into Java Class Library which is shipped together with the JRE, you don't need to specify the path to these classes in the CLASSPATH variable, they will be found automatically.

ℹ	**Note:** You should add the CLASSPATH variable to the system environment variables rather that to the user variables. While Lycia runs as local user and can utilize the user variables (which will allow you to compile your program), GUI clients run as administrator and if the CLASSPATH variable is set only for a user, a runtime error will occur.

Passing Options to the JVM

Java Virtual Machine is used to run java byte code integrated into 4GL. A JVM is distributed along with a set of standard class libraries that implement the Java application programming interface (API). Appropriate APIs bundled together form the Java Runtime Environment (JRE), so you need a JRE installed to be able to use Java interface.

If you compile your program in LyciaStudio and this program includes some Java classes references, LyciaStudio will look up the referenced classes in the Java packages located in the directories specified in CLASSPATH in the libraries shipped together with the JRE.

However, if you compile an application containing Java classes references in command line, you can instruct qfgl and qrun to pass command line options to the Java Virtual Machine during initialisation. This is achieved using the --java-option command line argument.

For example, you can pass the Java class path qfgl as follows:

```
$ qfgl --java-option=-Djava.class.path=<ClassPath> myprog.4gl
```

Note that the -cp and -classpath options supported by the Java runtime are not recognised when the Java Virtual Machine is initialised via the Java Native Interface; -Djava.class.path must be used instead.

You only need either the CLASSPATH variable set or the --java-option command line argument added to compile the application successfully. You do not need to set both. You also do not need to set any of them, if you use the classes from the standard Java Class Library.

Java Classes and Objects in 4GL

Using Java interface means using Java classes and Java objects as well as methods applicable to them within Lycia 4GL code. To be able to do this you must first import a Java class you want to use and then create the corresponding Java objects.

Importing a Java Class

To use a Java class within a 4GL program, you must import it. To import a java class the IMPORT JAVA statement is used. For a successful import the path to the Java class referenced should be specified either in the CLASSPATH variable or in the --java-option passed to qfgl, unless it is a standard Java class.

The syntax of the IMPORT JAVA is as follows:

```
IMPORT JAVA <ClassName>
```

For example to import the java.io.File class you should write:

```
IMPORT JAVA java.io.File
```

Note: The class name must be fully qualified when used in the IMPORT JAVA statement. It can be unqualified when referred to later in the program.

When referring to this class later in the program (e.g. in a variable declaration), either the fully qualified (e.g. java.io.File) or unqualified (e.g. File) variation may be used. The only exception to this rule is when the unqualified class name conflicts with a previously defined type, in which case, the fully qualified class name must be used throughout the program.

For each newly imported Java class you should use a separate IMPORT JAVA statement.

Scope of Reference of a Java Class

The IMPORT JAVA statement together with the FUNCTION, MAIN, REPORT, and GLOBALS statements can be used only outside any block (i.e. outside the MAIN block or outside the FUNCTION block). Thus, for example, you can use it before the MAIN block like this to be able to reference the class throughout the module - it will have the module scope of reference:

```
IMPORT JAVA java.util.regex.Pattern

MAIN

...

END MAIN
```

To import a Java class with the global scope of reference, you can add the IMPORT JAVA statement to the GLOBALS file. However, you cannot include this statement inside the GLOBALS statement - you should place it before the GLOBALS statement (e.g. like it is done for the DATABASE statement):

```
IMPORT JAVA java.io.File

GLOBALS

...

END GLOBALS
```

A Java class imported in a GLOBALS file can be referenced by any source file to which the GLOBALS file is linked.

Defining an Object Reference Variable

Before creating a Java object in Lycia 4GL, you must declare a program variable to reference the object. Before you will be able to declare such a variable, you need the corresponding Java class to be imported using the IMPORT JAVA statement first.

For example, you could declare a reference to a java.io.File object as follows:

```
IMPORT JAVA java.io.File

DEFINE f File
```

As previously noted, either the fully qualified or unqualified class name could be used in this declaration (assuming it does not conflict with a previously defined type). Here is an example of a fully qualified class name used for declaration:

```
IMPORT JAVA java.io.File

DEFINE p java.io.File
```

The scope of reference of this variable fully depends on the position of the DEFINE statement within the program and corresponds to that of any other 4GL variable in the given position.

Creating a Java Object

Objects can be created only for those classes which have a public constructor. Class instances cannot be created for classes which have only private constructors, e.g. an object cannot be created for the Pattern class.

To be able to call object methods of a Java class, you need to instantiate this class instance. To create an instance of a Java class, call the following method for a previously imported class:

```
<ClassName>.create()
```

The value returned should be assigned to an object reference variable of the appropriate type. For example:

```
IMPORT JAVA java.lang.StringBuffer

...

DEFINE sb StringBuffer

LET sb = StringBuffer.create()
```

Note: You cannot use a full-qualifier class name when invoking the *create()* method.

If the Java class constructor expects parameters, these should be passed to the *create()* method. In the example below the file name is passed as the parameter:

```
IMPORT JAVA java.io.File

...

DEFINE f File

LET f = File.create("filename")
```

You can create more than one object for the same class, if the class resources allow you to do so, e.g.:

```
IMPORT JAVA java.lang.StringBuffer

...

DEFINE sb1, sb2 StringBuffer

LET sb1 = StringBuffer.create()

LET sb2 = StringBuffer.create()
```

In this case these variables will reference two unique StringBuffer objects. The 4GL also allows the assignment of the value of one object reference variable to another, e.g.:

```
LET sb2 = sb1
```

In the above case the object referenced previously by variable *sb2* will be implicitly deleted and both variables will refer to one and the same object which was previously referenced only by *sb1*.

Calling Methods

Java interface allows you to call java methods from within a 4GL program. To able to call class methods you need to import the corresponding Java class and declare a variable referencing this class. To be able to call object methods, you need also create a Java object for the imported class.

Java interface also allows you to access the data stored in Java class and object fields and to pass them to 4GL variables.

Calling Class Methods

Class methods can be called without instantiating an object of the class. This way of calling a method is mainly used for the methods declared with the static keyword.

To call a class method the class should be previously imported using the IMPORT JAVA statement. You should also declare an object reference variable as described above. But you need not instantiate an object of this class.

To call a class method, the class name must be used as the prefix. If we use the class methods, we assign the result directly to the corresponding object reference variable.

```
<ClassName>.method()
```

For example:

```
IMPORT JAVA java.io.File

...

DEFINE f File

LET f = File.createTempFile("prefix", "suffix")
```

Calling Object Methods

Object methods can only be called once a class has been instantiated as an object and the object reference has been assigned to a variable. These methods are usually used for the methods without the static keyword.

To call an object method, the variable name must be used as the prefix.

```
<InstantiatedVariable>.method()
```

For example:

```
IMPORT JAVA java.io.File

...

DEFINE f File
DEFINE s STRING

LET f = File.create("filename") -- we instantiate the variable first

LET s = f.mkdir()
```

The result returned by such a method should be assigned to another 4GL variable.

Accessing Class Fields

The value of a class field can be read without instantiating an object of the class. Like with the class methods, you need to import the class and declare the corresponding variable.

To access a class field, the class name must be used as the prefix.

```
<ClassName>.ClassField()
```

For example:

```
IMPORT JAVA java.io.File

...

DEFINE s STRING

LET s = File.pathSeparator
```

	Note: In the current version of Lycia, in is not possible to change the value of a class field; only it's value may be read.

Accessing Object Fields

Object fields can only be read once a class instance has been instantiated, and the object reference has been assigned to a variable.

To access an object field, this variable name must be used as the prefix. The value returned must be assigned to a 4GL variable.

```
<InstantiatedVariable>.ObjectField()
```

For example:

```
IMPORT JAVA java.awt.Point

...

DEFINE p Point

DEFINE x_coord,y_coord INTEGER
```

```
# instantiate the object reference variable

LET p = Point.create(1,2)

LET x_coord = p.x -- access the field, returns 1

LET y_coord = p.y -- access the field, returns 2
```

In the example above p is the variable which contains the class object instantiated using the create(1,2) method - a point initialized at the given coordinates - 1,2. Also x in p.x and y in p.y are the names of the public fields of the Point class.

	Note: In the current version of Lycia, in is not possible to change the value of an object field; only it's value may be read.

Mapping Data Types

Lycia 4GL variables can be passed to Java methods and constructors as parameters. Java variables can also be passed to Lycia 4GL. Before the method or constructor is invoked, the parameters must be converted to values of the corresponding Java data types. This section describes how Lycia 4GL data types are mapped to Java data types and vice versa.

Method Overloading

The Java language allows method overloading; the parameter count and the parameter data types of a method are part of the method identification, thus several methods of one class can have the same name as long as they have different parameters.

The method or constructor that will be invoked at runtime is determined at compilation time using the data type mappings described in this chapter. In other words, make sure that, if you run the create() method or any Java method with 4GL variables as parameters, the data types of these parameters are mapped to the Java data types suitable for the method invocation.

Mapping Lycia 4GL Data Types to Java Data Types

When a 4GL variable is used as a parameter for calling a java method or in the create() method, you should bear in mind that 4GL data types need to be converted to Java data types before any further action is undertaken.

The data types are mapped according to the following table:

4GL Data Types	Java Data Types
CHAR **VARCHAR** **NCHAR** **NVARCHAR** **STRING** **TEXT**	java.lang.String
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
BOOLEAN	bool
SMALLFLOAT	float
FLOAT	double
DECIMAL **MONEY**	java.math.BigDecimal
DATE **DATETIME**	javax.xml.datatype.XMLGregorianCalendar
INTERVAL	javax.xml.datatype.Duration
BYTE	byte[]

Mapping Array Data Types

Lycia 4GL arrays (static or dynamic) can be passed to Java methods like any other program variable. The Lycia 4GL array will be converted to a Java array of the corresponding Java data type. If an array element is passed to a Java method, it is converted to a single parameter of the Java data type which corresponds to the data type of the element.

Mapping Record Data Types

Lycia 4GL records can be passed to Java methods like any other program variable. When passed to a Java method, the record will be converted into a Java object of the type java.util.HashMap.

The keys in the HashMap correspond to the names of the fields in the Lycia 4GL record. Fields of the Lycia 4GL record will be individually converted to Java data types using the conventions outlined in this section. Since the java.util.HashMap can only be used to contain Java objects, Lycia types that normally correspond to a Java primitive type (e.g. INTEGER, which maps to the Java int type) will be implicitly wrapped in a Java object of the appropriate wrapper class (e.g. java.lang.Integer).

If you pass an array of record to a Java method, it is converted into a Java array of HashMap objects.

A single record member passed to a Java method is converted into the Java data type corresponding to the data type of this member.

Mapping Java Data Types to Lycia 4GL Data Types

Java values can be returned to the Lycia 4GL program in response to the invocation of a Java method, or an attempt to read a Java field. In this case the returned values of Java data types need to be assigned to the 4GL variables of the mapped data types.

Mapping Primitive Data Types

The following Java types are converted immediately to the corresponding Lycia 4GL data types, either when they are returned by a Java method call, or the 4GL program reads a Java field of the given type:

Java Data Type	4GL Data Type
boolean	BOOLEAN
byte	TINYINT
char	INTEGER
short	SMALLINT
int	INTEGER
long	BIGINT
float	SMALLFLOAT
double	FLOAT

Java Object References

If a Java method returns a reference to a Java object, the value will remain as a Java object reference until it is assigned to a Lycia variable of a non-Java type. Valid assignments of Java data types to Lycia 4GL program variables are as follows:

Java Data Types	**Simple 4GL Data Types**
java.lang.String	CHAR VARCHAR NCHAR NVARCHAR STRING TEXT
java.lang.Byte	TINYINT
java.lang.Short	SMALLINT
java.lang.Integer	INTEGER
java.lang.Long	BIGINT
java.lang.Boolean	BOOLEAN
java.lang.Float	SMALLFLOAT
java.lang.Double	FLOAT
java.math.BigDecimal	DECIMAL MONEY
javax.xml.datatype.XMLGregorianCalendar	DATE DATETIME
javax.xml.datatype.Duration	INTERVAL
byte[]	BYTE
java.util.HashMap	RECORD *

***:** The keys in the HashMap must correspond to the names of the fields in the Lycia record. If they do not, a runtime error will occur.

Java Array References

Java arrays exist as subclasses of java.lang.Object. As such, they are handled much like any other Java object reference, until they are assigned to a Lycia 4GL program variable. A Java array reference may be assigned to a Lycia 4GL array provided the component type of the Java array can be mapped to the component type of the Lycia 4GL array. Individual elements of the array will be converted according to the conventions outlined in this section.

Operators

The implementation of Java interface requires the introduction of some new operators into Lycia 4GL.

The CAST Operator

Special attention must be paid when assigning object references to different target types or classes. There are two types of the Reference Conversions. Widening Reference Conversion is performed implicitly by the Lycia Java Interface. To perform Narrowing Reference Conversion, however, the CAST operator must be used. If the designated conversion is valid, this operator casts the left operand to an object reference of the class specified by the right operand. If the cast is invalid, a runtime error will occur.

```
CAST(<ObjectReference> AS <ClassName>)
```

The following example creates a java.lang.StringBuffer object, and assigns the reference to a java.lang.Object variable (implying Widening Reference Conversion); then the Object reference is assigned back to the StringBuffer variable (implying Narrowing Reference Conversion and CAST operator usage):

```
IMPORT JAVA java.lang.Object

IMPORT JAVA java.lang.StringBuffer

MAIN

 DEFINE o java.lang.Object

 DEFINE sb java.lang.StringBuffer

 LET sb = StringBuffer.create()

 LET o = sb                          -- Widening Reference Conversion

 LET sb = CAST(o AS StringBuffer)    -- Narrowing Reference

                                     -- Conversion needs CAST()

END MAIN
```

The INSTANCEOF Operator

The INSTANCEOF operator can be used to verify the class of an object. This operator checks whether the left operand is an instance of the class specified by the right operand:

```
<ObjectReference> INSTANCEOF <ClassName>
```

The example below creates a java.lang.StringBuffer object, assigns the reference to a java.lang.Object variable, and tests whether the class type of the object reference is a StringBuffer:

```
IMPORT JAVA java.lang.Object

IMPORT JAVA java.lang.StringBuffer

MAIN

 DEFINE o java.lang.Object

 LET o = StringBuffer.create()

 DISPLAY o INSTANCEOF StringBuffer    -- Shows 1 (TRUE)

END MAIN
```

Example

This chapter will illustrate the usage of a Java interface step by stem using a primitive sample of a Java class and a simple 4GL program referencing it.

A Sample Java Class

Here is the text of a Java class used for the purpose of demonstration. You should save it as a file called "SimpleClass.java" and compile into the file "SimpleClass.class". Then put the .class file into the location specified in your CLASSPATH variable.

```
public class SimpleClass {

    private int val;

    public SimpleClass() {

    }

    public SimpleClass(int val) {

        this.val = val;

    }

    public int getVal() {

        return val;

    }

    public void setVal(int val) {

        this.val = val;

    }

    public static int getValsSum(SimpleClass v1, SimpleClass v2) {

        return v1.getVal() + v2.getVal();

    }

}
```

Here we have the two public constructors either of which can be used for creating an instance of this class: SimpleClass() and SimpleClass(int val). Depending on whether you use the create() method with or without a parameter the corresponding constructor is invoked.

We also have a static method (a class method) which can be invoked without an object - getValsSum(), and a number of object methods with and without parameters which should be invoked using an instantiated class instance.

Importing and Using the Class

Now that you have the .class file and set up all the requirements, create a 4GL program in LyciaStudio:

```
IMPORT JAVA SimpleClass -- class is imported
MAIN
  DEFINE s, s1 SimpleClass,
       i, i1, param INT,
       ii BIGINT,
       ch CHAR

  PROMPT "Enter the parameter for SimpleClass object s: " FOR param

# constructor SimpleClass(int val) is invoked
  LET s = SimpleClass.create(param)

# constructor SimpleClass() is invoked
  LET s1=SimpleClass.create()

# these are object methods without parameters
  LET i=s.getVal()
  LET i1=s1.getVal()
  DISPLAY "The initial value of the object s  = ", i AT 3,2
  DISPLAY "The initial value of the object s1 = ", i1 AT 4,2

  INITIALIZE param TO NULL
  PROMPT "Enter the new value for SimpleClass object s1: " FOR param

# this is an object methods with parameters
  CALL s1.setVal(param)
  LET i1=s1.getVal()
  DISPLAY "The new value of the object s1 = ", i1 AT 4,2

# method getValSum is a class method, because it has 'static' keyword
  LET ii = SimpleClass.getValsSum(s,s1)
  DISPLAY "The getValPow() method returns: s1+s2 = ", ii AT 6,2

  PROMPT "Press any key to exit..." FOR CHAR ch
END MAIN
```

Here is the step-by-step explanation of the program:

1. Before the MAIN block include the following line to import the SimpleClass class:

   ```
   IMPORT JAVA SimpleClass
   ```

   This class is not included into any package, thus it has no qualifiers. If an imported class has qualifiers, they should be specified here.
2. Inside the MAIN section, define two variables pointing at this class:

   ```
   DEFINE s, s1 SimpleClass
   ```

3. Create an object of this class for each variable. Thus you will have two independent objects. They will not be identical, because the create() method invokes different constructors due to different parameters.
   a. For variable *s* the object is created with the parameter received from the user. 4GL variable *param* is of INTEGER data type, so it invokes SimpleClass(int val) constructor which expects Java int value as a parameter. 4GL INTEGER and Java int data types are mapped, so the object should be created without problems. This object will contain the value passed to it.

      ```
      LET s = SimpleClass.create(param)
      ```

   b. For variable *s1* the object is created without any parameters. So the create() method invokes SimpleClass() constructor which does not expects any parameters. This object will contain no value (0) initially.

      ```
      LET s = SimpleClass.create(param)
      ```

4. Then we check the values of the objects using an object method on both object variables.

   ```
   LET i=s.getVal()

   LET i1=s1.getVal()
   ```

5. After that we offer the user to enter the new value for object s1 (which was initialized with 0 value). This method contains the void keyword, so it does not return anything. Thus it is just called without the receiving variable.

   ```
   LET s1 = s1.setVal(param)
   ```

6. When we use the getVal() method again, we will see the changed value returned.
7. The next method used is a class method. It accepts the two objects as parameters, in our case these are s and s1. It must be called with the class name as the prefix. If you try to call this method as an object method, you will get a runtime error produced by the JVM. The value returned by the method is of Java long data type, thus it is assigned to the corresponding variable of the BIGINT data type for compatibility. However, if the value is short enough it can be assigned to INT, SMALLINT or TINYINT variables instead.

   ```
   LET ii = SimpleClass.getValsSum(s,s1)
   ```