Calling Java code from Biferno

The jclass Extension

Valerio Ferrucci, Tabasoft Sas

Table of Contents

1. jclass Extension . . *1*
    1.1. Versions and Requirements . . *1*
    1.2. Installation . . *2*
    1.3. Locating the JVM library . . *2*
    1.4. Compiling from Source . . *3*
2. Loading and using a Java Class . . *4*
3. Prototype Detection . . *6*
    3.1. Typecast . . *7*
4. Examples . . *8*
    4.1. Matcher and Pattern . . *8*
    4.2. Reflection and arrays . . *8*
5. Properties, Public, Private and Protected Members . . *9*
6. The jclass null Property . . *9*
7. Calling User Classes . . *10*
8. JVM Cache and User Classes . . *12*
9. jclass Constructor Complete Prototype . . *13*
10. System.out, System.err and errors . . *14*
References . . *14*

Abstract

The Biferno *jclass* extension, developed by Tabasoft, allows Biferno code to directly invoke Java classes. Developers can now take advantage of any Java 2 Platform classes in their Biferno web sites. Besides, if they have developed their own Java classes, they can reuse that code in `bfr` scripts with no modifications. [1]

1. jclass Extension

1.1. Versions and Requirements

The first version of *jclass* available to developers is 1.1 and is packaged together with Biferno 1.1 (usually Tabasoft extensions have the same versions of the package they are distributed with). At least Biferno version 1.1 is required to use the `jclass` extension.

The extension is available to Linux, Windows and MacOSX developers. A MacOS classic version will not be delivered (note that Java availability on MacOS classic has stopped at version 1.1.8, implemented in MRJ 2.2.5, while `jclass` requires at least Java 2).

To use `jclass`, you will need a machine (running Linux, Windows or MacOSX) with the following installed:

- Biferno version >= 1.1

[1] Java® and all Java-based marks are a trademark or registered trademark of Sun Microsystems, Inc, in the United States and other countries.

- Java 2 SDK (at the time of this writing the current version is: *j2sdk1.4.2*)

1.2. Installation

The `jclass` extension must be in the `Extensions` folder of `BifernoHome`. The exact location depends on the platform:

1. Linux: `/home/BifernoHome/Extensions/jclass_bfr.so`. The Biferno (>= 1.1) rpm installer/updater copies it to the right location and no additional operations are needed.
2. MacOSX: `/Users/BifernoHome/Extensions/jclass_bfr.so`. The Biferno (>= 1.1) installer/updater copies it to the right location and no additional operations are needed.
3. Windows: `[Program Files]/Biferno/BifernoHome/Extensions/jclass_bfr.dll`. The Biferno (>= 1.1) installer/updater copies it to the right location and no additional operations are needed.

If the jclass is installed while Biferno is running, a complete reload of Biferno is needed to cause the loading of the new extension. Instructions to do a complete (from console or `BifernoCtl`) reload of Biferno can be found in [BFRIA].

1.3. Locating the JVM library

When the `jclass` is loaded, it needs to locate the JVM (Java Virtual Machine) libraries, for such libraries contain the code that actually loads the Java Virtual Machine into Biferno.

If `jclass` can't find the Java Virtual Machine, the extension is aborted (nevertheless Biferno startup continues regularly). If you discover such an abort in the Biferno startup log (i.e. on Windows you see the message: "The specified module could not be found" when trying lo load the `jclass` extension) please read carefully the following notes.

The name and location of the JVM libraries depend on the platform:

- Linux:

  the name of the library is `libjvm.so`. If the library is not in a standard location (`/usr/lib` etc...) you can tell the system loader where to find it by setting or modifying the `LD_LIBRARY_PATH` environment variable. For example on my machine the jvm library is located in:

  `/usr/java/j2sdk1.4.1_02/jre/lib/i386/server/libjvm.so`

  so I need to use the following command in my `bifernodefs.sh` script file (`/etc/rc.d/init.d/bifernodefs.sh`):

  ```
  export LD_LIBRARY_PATH= $LD_LIBRARY_PATH:/usr/java/\
  j2sdk1.4.1_02/jre/lib/i386/server/:/usr/java/j2sdk1.4.1_02/\
  jre/lib/i386/
  ```

  (note that the '\' char is there only to make the shell ignore the ignore the carriage return when running the script). The last path in `LD_LIBRARY_PATH` was needed because `libjvm.so` searches for other libraries in that location (see "Defining Environment Variables for Biferno" in [BFRIA] for details on the `bifernodefs.sh` file).

  If you change environment variables in the Biferno startup script, relaunch Biferno using the command:

  ```
  sudo /etc/rc.d/init.d/biferno restart
  ```

- MacOSX:

  the name of the library is `libjvm.dylib`. You can tell the system loader where to find it modifying the `LD_LIBRARY_PATH` (or `DYLD_LIBRARY_PATH`) environment variable. For example on my machine I have the jvm library in:

  ```
  /System/Library/Frameworks/JavaVM.framework/Versions/1.4.2/Libraries/libjvm.d
  ```

  so I put the following command in my `bifernodefs.sh` script (`/Library/StartupItems/Biferno/bifernodefs.sh`):

  ```
  export LD_LIBRARY_PATH=/System/Library/Frameworks/\
  JavaVM.framework/Versions/CurrentJDK/Libraries
  ```

  (note that the char '\' is only there to make the shell ignore the ignore the carriage return when running the script). `CurrentJDK` is a symbolic link to the current version installed on my machine (as of today in the `1.4.2` folder).

  See "Defining Environment Variables for Biferno" in [BFRIA] for details on the `bifernodefs.sh` file.

  If you change environment variables in the Biferno startup script, quit Biferno and relaunch it using the commands:

  sudo bifernoctl stop

  sudo /Library/StartupItems/Biferno/Biferno

- Windows:

  the name of the library is `jvm.dll`. You can tell the system loader where to find it by accessing the Control Panel section:

  `System->Advanced->Environment Variables...`

  and modifying the `Path` System variable. For example on my Win2000 machine I have the jvm library in

  `D:\Program Files\j2sdk_nb\j2sdk1.4.2\jre\bin\client\jvm.dll`

  so I modified the `Path` var to contain the values:

  `D:\Program Files\j2sdk_nb\j2sdk1.4.2\jre\bin\client;D:\Program Files\j2sdk_nb\j2sdk1.4.2\bin`

  (don't insert a carriage return in the line). Note that on Windows the separator between paths is the '`;`' character (on Unix it is '`:`'). The second path (after the '`;`') contains other dll's needed by the jvm lib. Note also that the `Path` variable is needed by many applications on your computer, so you have to add the jvm paths without deleting what is already present in it.

  If you changed environment variables, quit Biferno and relaunch it using the BifernoCtl application (see [BFRIA] for details).

1.4. Compiling from Source

This section gives useful information to developers who want to recompile the `jclass` Biferno extension from source. If you just want to use the binary extension, skip this section.

- *Linux* and *MacOSX*:

The `Makefile` is located in: `External/jclass/Makefile`. The include headers are searched for in these folders:

```
${JAVA_HOME}/include
${JAVA_HOME}/include/linux
```

(the second path only refers to Linux). So the `JAVA_HOME` environment variable must be defined and must evaluate to the location of the JAVA main folder (i.e. `/usr/java/j2sdk1.4.1_02`).

No libraries are linked at compile time. They are loaded at runtime as explained in Section 1.3, “Locating the JVM library”.

- *Win32*:

  The `jclass` extension is not compiled by default in the `Biferno.dsw` workspace. To compile it open the `External/jclass/jclassVC/jclassVC.dsw` project and build the extension.

  The include search paths to add to your environment (in tool->options->directories) are:

```
[path to J2SDK]\J2SDK[version]\INCLUDE
[path to J2SDK]\J2SDK[version]\INCLUDE\WIN32
```

  for example on my system the paths are:

```
D:\PROGRAM FILES\J2SDK_NB\J2SDK1.4.2\INCLUDE
D:\PROGRAM FILES\J2SDK_NB\J2SDK1.4.2\INCLUDE\WIN32
```

  No libraries are linked at compile time. They are loaded at runtime as explained in Section 1.3, “Locating the JVM library”.

2. Loading and using a Java Class

If the installation and loading of `jclass` was successful, you are ready to load your first Java class into Biferno. Let's start with a simple example.

Suppose we want to use the `java.util.Hashtable` class in our `bfr` script (Biferno does not have a standard class for hash tables, nevertheless it would not be difficult to implement one in bfr code, but let's use the one supplied by Java as an example).

The first step is to connect to the class using the `jclass` constructor in its simplest form:

```
<?biferno
Hashtable = jclass("java.util.Hashtable")
?>
```

this command creates a biferno variable of class `jclass` that points to the `java.util.Hashtable` Java class. Note that we named the variable using the same name as the class. Although I usually name variables with an initial lower case character, in this case I like to use the same name of the corresponding Java class. In fact it can be useful when referencing the class in the subsequent code (as we will see later). Note also that the fully qualified name of the Java class is needed: (`java.util.Hashtable`).

We can then instantiate a Hashtable object using the jclass method *new*.

```
<?biferno
myHT = Hashtable.new()
?>
```

Now `myHT` is a Biferno object that references a Java object of class Hashtable.

The `jclass` method `new` invokes the constructor of the class. It is the only method explicitly declared by jclass but, as we will see further, every Java method can be called on a `jclass` object.

Suspending the execution of the script with the Biferno `debug` command can help us understanding what is happening:

Figure 1. jclass variables in the error page

Local			
Name	**Type**	**Class**	**Value**
Hashtable	var	jclass	[java.util.Hashtable]
myHT	var	jclass	{} [java.util.Hashtable]

`Hashtable` is a `jclass` variable pointing to the `java.util.Hashtable` class and its string representation is just the complete name of the Java class.

`myHT` is a `jclass` variable referencing a Java object and its string representation is the result of the `toString()` Java method called upon the object (remember that all Java objects have such a method, either because they declare it or because they inherit it from the `java.lang.Object` class).

As we have just seen, there are two different types of jclass objects:

- References to classes (as `Hashtable` in our example) are called also *connectors*.
- References to objects (as `myHT` in our example) obtained calling the `new` method on a connector or, as we will see below, calling a Java method that returns a non-primitive Java object.

We can then add some values to our Hashtable by just applying the `put` method (declared in `java.util.Hashtable`) on our `myHT` variable:

```
<?biferno
myHT.put("John", "Lennon");
myHT.put("George", "Harrison");
myHT.put("Ringo", "Star");
myHT.put("Paul", "McCartney");
?>
```

`jclass` doesn't declare a `put` method but it recognizes that it is a method applicable to `myHT` and executes it (this mechanism is called *dynamic member declaration* in Biferno).

Now the object pointed to by `myHT` also contains the four items just added, as we can see by suspending the execution of the script with the Biferno `debug` command:

Figure 2. The HashTable Filled

Local			
Name	**Type**	**Class**	**Value**
Hashtable	var	jclass	[java.util.Hashtable]
myHT	var	jclass	{Ringo=Star, George=Harrison, John=Lennon, Paul=McCartney} [java.util.Hashtable]

3. Prototype Detection

As Java developers already know, the Java Language Specification states that methods and constructors in the same class can have the same name if they have different prototypes. For example the `Hashtable` Java class declares the following constructors:

```
Hashtable()
Hashtable(int initialCapacity)
Hashtable(int initialCapacity,  float loadFactor)
Hashtable(Map t)
```

How does `jclass` determine which constructor to invoke when the user writes:`myHT = Hashtable.new()`?

`jclass` constructs a prototype starting from the parameter passed by the user and looks for a method with a “compatible” prototype. In the example above no parameters were passed by the user, so `jclass` searches, in `java.util.Hashtable`, for a constructor with zero parameters, and finds one.

Let's see a more complex example. Suppose that you want to call the third constructor (`Hashtable(int initialCapacity, float loadFactor)`). In this case you would use a code like this:

```
<?biferno
myHT = Hashtable.new(4, 4)
?>
```

but this will cause the `Err_NoSuchMethod` error. The reason is that the literal `4` is translated by Biferno in a Java `int` and no constructor exists with prototype

```
Hashtable(int, int)
```

The following table reports how Biferno translates its primitive types to Java primitives:

Table 1. Biferno and Java Primitives

Biferno	Java
boolean	boolean
string	java.lang.String
double	double
long	long
unsigned	unsigned
int	int
char	char

Biferno	Java
array	array

Java has more primitives than Biferno. To address this issue `jclass`, in addition to declaring the "jclass" class itself, declares classes that complete the table above:

Table 2. Classes Declared by jclass (Other Than "jclass" Itself)

Biferno	Java
byte	byte
short	short
float	float

Biferno `byte` and `short` are the same as `int`, and `float` is the same as `double`. They are declared for the sole purpose of having the same primitives as Java available for prototype detection.

Now let us change the above example to:

```
<?biferno
myHT = Hashtable.new(4, float(4))
?>
```

The code now is correct and works as expected because a constructor with the requested prototype is found.

Note that writing:

```
<?biferno
myHT = Hashtable.new(4, 4.0)    // causes Err_NoSuchMethod
?>
```

is not correct because `4.0` is mapped by Biferno to `double`, not to `float`.

All we have said in this section about constructors applies in the same way to all class methods.

3.1. Typecast

In the previous section we stated that the prototype built from passed parameters must be "compatible" with that of an existent Java method. That is, every parameter class must be "compatible" with the corresponding parameter class in the searched Java method.

As an example we can use the `put` method we used in the previous example:

```
<?biferno
myHT.put("John", "Lennon");
myHT.put("George", "Harrison");
myHT.put("Ringo", "Star");
myHT.put("Paul", "McCartney");
?>
```

The prototype of `put` (as declared in Hashtable) is:

```
put(Object key,  Object value)
```

but we called it using `String` (Biferno `string`) instead of `Object`. The code is OK because the `String` class can be typecasted by Java to a generic `Object` (all classes are extensions of `Object`).

In particular Biferno uses the Java method:

```
public boolean isAssignableFrom(Class cls)
```

of the Java `Class` class to check if two classes are compatible and can be considered the same class in the prototype detection phase.

4. Examples

4.1. Matcher and Pattern

In the following example we use the `java.util.regex.Pattern` class to substitute strings within a text:

```
<?biferno
// connect to Pattern class
Pattern = jclass("java.util.regex.Pattern")
// Initialize our text
text = "today is Sunday"
// call the static method compile that return a Pattern variable
p = Pattern.compile("\\bSunday\\b")
// the matcher method return a java.util.regex.Matcher
m = p.matcher(text)
// then we can apply the replaceAll on the Matcher
result = m.replaceAll("Monday")
$result
?>
```

the output of the example is:

```
today is Monday
```

We can see that:

- Java static methods must be applied to connectors, not to instances, as in: `p = Pattern.compile("\\bSunday\\b")`.
- If the return value of a method is an object, Biferno automatically creates a `jclass` variable pointing to the new object, result of the method, as in `p = Pattern.compile("\\bSunday\\b")`. We can ask the class of that object using the command: `p.getClass()`. Primitive return values are mapped by Biferno into its primitive types, so no new `jclass` variables are allocated.
- The prototype of `matcher` (as declared in `java.util.regex.Pattern`) is:

  ```
  Matcher matcher(CharSequence input)
  ```

  As we passed a string to the method and no error is returned, we desume that Java `CharSequence` and `String` are compatible (`CharSequence` is really an *Interface* but discussing interfaces goes beyond the scope of this article, see [JUSTJ] for details about *Java Interfaces*).

4.2. Reflection and arrays

In the following example we explore the `BigDecimal` Java class using Java reflection. In particular we will list all the methods declared by the class `BigDecimal`.

```
<?biferno
// connect to the class java.lang.Class
Class = jclass("java.lang.Class")
// call the static method "forName" to get info about BigDecimal
bdClass = Class.forName("java.math.BigDecimal")
// get the array of methods
methods = bdClass.getMethods()
```

```
// display the methods
totMethods = methods.dim
for (i = 1; i <= totMethods; i++)
        $methods[i] + "<br>"
?>
```

The output of the script is very long and I report here only the beginning:

```
public java.math.BigInteger java.math.BigDecimal.unscaledValue() ... (continues
```

From the example we desume that:

- It is not necessary to call the constructor (`jclass.new`) to have a new `jclass` object. In fact a method call can return a new object, as in: `bdClass = Class.forName("java.math.BigDecimal")`.
- In the example the Java array returned by `bdClass.getMethods()` is automatically mapped to a Biferno array. In particular a Java array of `java.lang.reflect.Method` objects is mapped to a Biferno array of jclass object (each one pointing to a `java.lang.reflect.Method` Java object)
- Since the array returned by `bdClass.getMethods()` is a Biferno array, a `for` command can just loop on it printing each element. The print (`$`) command obtains a string implicitly calling the Java method `toString()` of class `java.lang.reflect.Method`.

5. Properties, Public, Private and Protected Members

Properties of classes can be accessed (and modified). The same rules explained for methods are valid for properties too.

Note however that members (properties and methods) of Java classes can be invoked from Biferno code only if they are public. Protected and Private members are not available to Biferno code and cannot be accessed or modified.

If a non-public member is accessed the error `Err_CantAccessThisMember` is thrown.

For example consider the following code

```
<?biferno
// connect to the class java.lang.Class
Class = jclass("java.lang.Class")
// call the static method "forName" to begin reflection on BigDecimal
bdClass = Class.forName("java.math.BigDecimal")
// get the class loader of BigDecimal
cLoader = bdClass.getClassLoader()
// get the parent of the class loader
$cLoader.parent
?>
```

This code throws the `Err_CantAccessThisMember` error because the `parent` property is private to the `ClassLoader` class (instead, the `getParent()` method should be used).

For details about Java *public*, *private*, and *protected* members see [JUSTJ] (if you are interested in Java ClassLoaders see [JVCLD] or [UNCFN]).

6. The jclass `null` Property

Let's modify the previous example to try to obtain the correct ClassLoader of `BigDecimal` (and its

parent). We use the `getParent()` method instead of the `parent` property.

```
<?biferno
// connect to the class java.lang.Class
Class = jclass("java.lang.Class")
// call the static method "forName" to begin reflection on BigDecimal
bdClass = Class.forName("java.math.BigDecimal")
// get the class loader of the class
cLoader = bdClass.getClassLoader()
// get the parent of the class loader
$cLoader.getParent()
?>
```

We discover that this code throws a `Err_JClassError` error with message *Error: can't apply method to null* at the line: `cLoader.getParent()`. What is happening is that the class loader for `BigDecimal` (`cLoader`) is just `null` (you can discover why this is the case by studying how Java class loaders work).

`null` is a special Java object. It is an object of a certain class but with value `0`. Biferno does not have the concept of `null`, so if you want to create a Java null object to use in your Biferno code you have to use the `jclass` property:

```
jclass null
```

For example, to obtain a Java null object of class `String` use the code:

```
<?biferno String = jclass("java.lang.String") nullString = String.null ?>
```

If the above script is interrupted by a debug command we see the following local variables list:

Figure 3. null jclass variable example

Local			
Name	**Type**	**Class**	**Value**
String	var	jclass	[java.lang.String]
nullString	var	jclass	null [java.lang.String]

7. Calling User Classes

User classes can be called by Biferno in the same way as Java Platform classes.

As an example we build the class HelloWorld from the Java code:

```
class HelloWorld
{
    public static void main(String[] args)
        {
        SayHello();
        }

    public static void SayHello()
        {
        System.out.println("Hello World!");
```

```
        }
}
```

put the code in a file called `HelloWorld.java` then type from the console:

javac HelloWorld.java

java HelloWorld

and you should see on the console the output:

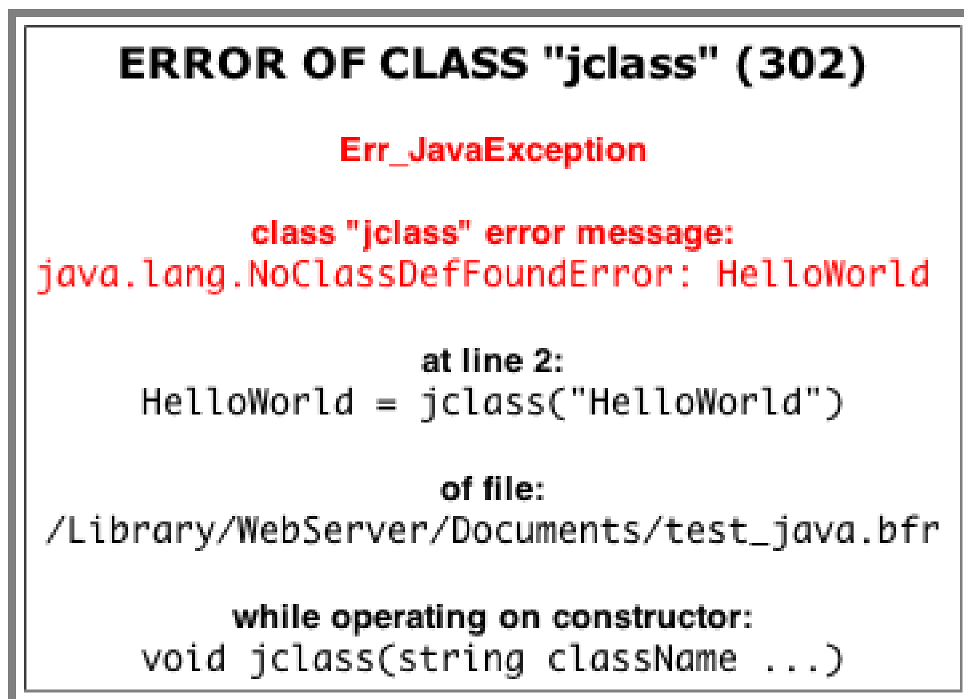
Hello World!

Now, we try to call the method `SayHello()` from Biferno:

```
<?biferno
// connect to HelloWorld class
HelloWorld = jclass("HelloWorld")
// call the static method SayHello
HelloWorld.SayHello()
?>
```

Running the biferno code we get the error:

Figure 4. User class error



it looks like Biferno (better, the JVM inside Biferno) doesn't find my class. Really, that should be no surprise as I didn't say where to look for it. To fix the problem we have to tell Biferno where my classes are located. This can be done by setting the `CLASSPATH` environment variable. On my machine I added the following line to the Biferno startup script:

```
CLASSPATH=/Library/WebServer/Documents
```

and restarted Biferno (refer to section Section 1.3, "Locating the JVM library" to learn about setting environment variables for Biferno on different platforms).

After the restart of Biferno my script returns no error and shows the following output:

```
Hello World!
```

In the startup log of Biferno we can find the value of `CLASSPATH` as it is recognized by Biferno. If it doesn't correspond to our classes location, check the value of the environment variable and be sure that the Biferno process can see that value. If `CLASSPATH` contains many paths, they must be separated by the'`:`' character (or '`;`' on Windows) .

8. JVM Cache and User Classes

Because the process of loading a class from disk to memory is slow, the JVM does it only the first time the class is invoked and then keeps the class in memory (in the JVM *cache*) for successive executions. The JVM loaded by Biferno has this mechanism too, allowing Java methods included in a Biferno script to be fast and efficient.

Nevertheless, the JVM cache can be an annoying effect when we are developing a class because every modification (and recompilation) of the class doesn't affect the copy already loaded in Biferno.

As an example, consider the following modification to the source of the previous example:

```
class HelloWorld
{
    public static void main(String[] args)
        {
        SayHello();
        }

    public static void SayHello()
        {
        System.out.println("Hello good World!");
        }
}
```

(I added “good” in the printed text). Type from the console:

javac HelloWorld.java

java HelloWorld

and you should see on the console the new output:

Hello good World!

Executing the Biferno script, instead, you will continue to see:

```
Hello World!
```

This clearly is a problem and we need a fix for this (the brute force approach would be to restart Biferno every time a modification is made, but this is very slow and painful!).

To work around this the `jclassExt` class must be used instead of `jclass` during the development phase. `jclassExt` is a class that uses a special ClassLoader that causes the classes *not to be cached* by the JVM.

Tip

The JVM removes a class from memory when the class loader that loaded it is garbaged (exits from its scope). Normally the class loader used by Biferno is the System loader, that is never garbaged. The `jclassExt` class simply doesn't use the System class loader, but another one that exits from scope at every biferno script execution. A complete explanation of the Java Class Loaders would be very long here and is beyond the scope of this article, if you are interested, you can see [JVCLD] or

[UNCFN].

jclassExt is distributed in the Utils folder of BifernoHome and, to work properly, it needs the BfrClassLoader class loader. The latter is a class loader implemented by Tabasoft whose .java and .class files are in BifernoHome. **Copy the file BifernoHome/BfrClassLoader.class to your CLASSPATH folder** and modify the script source to read:

```
<?biferno
include(biferno.home + "Utils/jclassExt.bfr")
// connect to HelloWorld class using jclassExt
HelloWorld = jclassExt("HelloWorld")
// call the static method SayHello HelloWorld.SayHello()
?>
```

Note that, at this point, we need to restart Biferno because the class was already loaded in memory. Restart Biferno and relaunch the script. From now on we will see all the modification to our Java code just reflected in the Biferno script output.

Don't forget to replace jclassExt with jclass when the debugging phase is finished because the JVM cache usually strongly improves the performance of your bfr/java pages.

9. jclass Constructor Complete Prototype

It can be interesting to read the jclassExt implementation source file (it is in BifernoHome/Utils/jclassExt.bfr). In particular the constructor:

```
<?biferno
...
void jclassExt(string className)
{
	if (biferno.os.Contains("UNIX"))
		sep = ":"
	else
		sep = ";"
	ar = unix.getenv("CLASSPATH").ToArray(sep)
	tot = ar.dim
	name = "/" + className + ".class"
	for (i = 1; i <= tot; i++)
	{	classFilePath = file.BifernoPath(ar[i]) + name
		if (file.Exists(classFilePath))
			break
	}
	BfrClassLoader = jclass("BfrClassLoader")
	cl = BfrClassLoader.new()
	super(className, cl, classFilePath)
}
...
?>
```

we can see that it calls the jclass constructor (super) with three parameters. Really the prototype of jclass constructor is declared as:

```
jclass(string className...)
```

and it can be considered as:

```
jclass(string className, jclass classLoader, string classLoaderFilePath)
```

with the second and third parameter always absent. If passed, the parameters after the first one are interpreted as:

- A jclass variable referencing a Java `ClassLoader` object
- The path of the `.class` file containing the implementation of the class `className` (path follows Biferno conventions)

In this way the *canonical* (*System*) Class Loader can be substituted by a custom Class Loader (and this is what `jclassExt` does to work around the cache problem).

10. System.out, System.err and errors

In our previous example, the Java code executes the command:

```
System.out.println("Hello World!");
```

and as a result, when the method is invoked by Biferno, we see the string “Hello World!” in our browser window. In fact Biferno redirects standard output to the output of the `bfr` script (that is, it adds the text to the variable `global pageOut.body`).

In particular Biferno redirect all the calls to:

```
System.out
```

to the bfr script output and the calls to

```
System.err
```

to the Biferno error page `jclass` message. In this way we can have always a good description of the errors and a stack trace is displayed to know which method caused the error and the callers of the method itself.

`jclass` can throws the following class errors:

- *Err_JavaClassNotFound*: this error is thrown when a Java class having the name specified by the user can't be found. Usually this error is thrown by the `jclass` constructor but it can also arise from method calls.
- *Err_JavaException*: an exception was thrown by the Java code. The `jclass` message in the Biferno error page gives more details about the exception.
- *Err_JavaFieldNotFound*: an internal error happened while getting the value of a class field (property). It should never happen.
- *Err_JavaStringTooLong*: a Java string that was too long caused an error (this should never happen).
- *Err_JClassError*: a generic `jclass` error occurred. The `jclass` error message in the Biferno error page provides more details about the error.

References

[BFRLG] Biferno Language Guide Tabasoft S.a.s.

[BFRIA] Biferno Installation and Administration Guide Tabasoft S.a.s.

[JUSTJ] Just Java Peter van der Linden Sun Microsystems

[JVNUT] Java in a Nutshell David Flanagan O'Reilly & Associates, Inc.

[JVCLD] The basics of Java class loaders The fundamentals of this key component of the Java architecture Chuck McManis http://www.javaworld.com/javaworld/jw-10-1996/jw-10-indepth.html

[UNCFN] Understanding Class.forName() Loading Classes Dynamically from within Extensions Ted Neward http://www.javageeks.com/Papers/ClassForName/index.html

[JNI] Essential JNI Java Nativa Interface Rob Gordon Prentice Hall PTR