The JavaToDPR 1.0.3 Update

by Daniel U. Thibault
D.U.Thibault@Bigfoot.com

Prefatory comment: All of what follows was developed and tested under Java 2 1.4.2.

The original `JavaToDPR` class is very usefull, but it does break under certain conditions. Here we will take a closer look at three issues and how they affect the behaviour of JNI through `javah` (which `JavaToDPR` is supposed to mimic):

1) Inner classes
2) Embedded underscores and dollars in identifiers
3) Overloaded native methods

Here is a sample plain class:

```
public class
samplePlainClass
{
   native public int
   samplePlainNativeMethod();
}
```

It uses the default contructor (which we'll do throughout this, as constructors cannot be native and thus don't affect JNI-Delphi much), and declares a single plain native method.

Running `javah samplePlainClass` gets us the `samplePlainClass.h` file:

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

#ifndef _Included_samplePlainClass
#define _Included_samplePlainClass
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     samplePlainClass
 * Method:    samplePlainNativeMethod
 * Signature: ()I
 */
JNIEXPORT jint JNICALL Java_samplePlainClass_samplePlainNativeMethod
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

From now on, we will omit the black lines, which don't change or matter.

No surprises here. The corresponding DPR (obtained by invoking `java JavaToDPR -o <dpr name>.dpr -platform Win32 samplePlainClass`) is:

```
library samplePlainClass;

uses JNI;

(*
 * Class:      samplePlainClass
 * Method:     samplePlainNativeMethod
 * Signature:  ()I
*)
function Java_samplePlainClass_samplePlainNativeMethod(PEnv: PJNIEnv;
Obj: JObject): JInt; stdcall;
begin
end;

exports
  Java_samplePlainClass_samplePlainNativeMethod;

end.
```

The correspondences are:

DPR	H
library name	(file title)
Comments	Comments
Method signature	Translation of method signature
Exports clause	Repeats method signature

As expected, the JNI method signature consists of:
"`Java_<class name>_<method name>`".

Introducing packages

Of course, most classes won't lie in the default package. Here we move the `samplePlainClass` to the `samplePackage`. This is done by adding a line to the source file:

```
package samplePackage;

public class
samplePlainClass
{
   native public int
   samplePlainNativeMethod();
}
```

Here we run into an oddity. Running `javah samplePlainClass` works and gets us the exact same file. However, we should really run `javah samplePackage.samplePlainClass`, which gets us these slight changes:

```
/* Header for class samplePackage_samplePlainClass */
/*
 * Class:     samplePackage_samplePlainClass
 * Method:    samplePlainNativeMethod
 * Signature: ()I
 */
JNIEXPORT jint JNICALL
Java_samplePackage_samplePlainClass_samplePlainNativeMethod
  (JNIEnv *, jobject);
```

Note how the dots (".") used in Java code to denote the package path (our class' name is now `samplePackage.samplePlainClass`) change to underscores. The JNI method signature now consists of:
"`Java_<package path>_<class name>_<method name>`".

The DPR (obtained by invoking `java JavaToDPR -o <dpr name>.dpr -platform Win32 samplePackage.samplePlainClass`) follows suit:

```
library samplePackage_samplePlainClass;
(*
 * Class:      samplePackage_samplePlainClass
 * Method:     samplePlainNativeMethod
 * Signature:  ()I
*)
function
Java_samplePackage_samplePlainClass_samplePlainNativeMethod(PEnv:
PJNIEnv; Obj: JObject): JInt; stdcall;
exports
  Java_samplePackage_samplePlainClass_samplePlainNativeMethod;
```

Introducing inner classes

Inner classes are a powerful feature of the Java language. Here we won't consider anonymous inner classes but concentrate on named inner classes instead. These classes are declared within the declaration of an enclosing class, and their instances can exist only in relation to an enclosing instance. In particular, instead of writing:

```
samplePlainClass sPC = new samplePlainClass();
```

If we have an inner class `samplePlainInnerClass`, we would write:

```
samplePlainInnerClass sPIC = sPC.new samplePlainInnerClass();
```

Should you need to locate an inner class' constructor through the JNI, note that the ObjectMethod name is "`<init>`" and the signature "`(<enclosing class signature><remaining constructor signature>)V`". For example, for our `samplePlainInnerClass`, the (default) constructor signature is "`(LsamplePackage/samplePlainClass;)V`".

Things get even more interesting if one wants to extend an inner class from outside of the enclosing class, but that is beyond the scope of this document.

We add `samplePlainInnerClass` to our `samplePlainClass` like so:

```
package samplePackage;

public class
samplePlainClass
{
   native public int
   samplePlainNativeMethod();

   public class
   samplePlainInnerClass
   {
      native public float
      samplePlainNativeInnerClassMethod();
   }
}
```

Compiling this creates the usual `samplePlainClass.class` file, but also a `samplePlainClass$samplePlainInnerClass.class` file. Indeed, the inner class' full name is `samplePackage.samplePlainClass$samplePlainInnerClass`.

We must run `javah` separately on the two classes. The outer class' header does not change. The inner class' header is obtained by running `javah samplePackage.samplePlainClass$samplePlainInnerClass` and comes out like this (`samplePackage_samplePlainClass_samplePlainInnerClass.h`):

```
/* Header for class samplePackage_samplePlainClass_samplePlainInnerClass
*/
/*
 * Class:     samplePackage_samplePlainClass_samplePlainInnerClass
 * Method:    samplePlainNativeInnerClassMethod
 * Signature: ()F
 */
JNIEXPORT jint JNICALL
Java_samplePackage_samplePlainClass_00024samplePlainInnerClass_samplePla
inNativeInnerClassMethod
  (JNIEnv *, jobject);
```

We note several things. First, the periods and dollars are treated the same for labeling the header file. Second, we see a divergence between the comment declaring the class name (same as the file title) and the actual native method declaration. The JNI method signature now consists of:
"`Java_<package path>_<class name>_00024<inner class name>_<method name>`".
The dollar is represented by "`_00024`". An inner class could contain an inner class, and so on, leading to a chain of dollar-separated names.

JavaToDPR 1.0.0 breaks at this point, being unable to reproduce the inner class method signatures correctly.

Introducing underscores and dollars

Java identifiers are very flexible, being able to use an overwhelming range of Unicode characters. Interesting things happen (or fail to happen) when this spreads to the file system. Here, we will concentrate on the two characters that the Java Language Specification singles out as additional Java letters: the ASCII underscore (Unicode \u005f) and the dollar sign (\u0024). We will introduce them both in the package, class and method names, just to be really pernicious:

```
package sample_$tricky;

public class
sample_$trickyClass
{
   native public int
   sample_$tricky_NativeMethod();

   public class
   sample_$tricky_InnerClass
   {
      native public float
      sample_$tricky_NativeInnerClassMethod();
   }
}
```

Running `javah sample_$tricky.sample_$trickyClass` yields `sample_0005f_tricky_sample_0005f_trickyClass.h`:

```
/* Header for class sample_0005f_tricky_sample_0005f_trickyClass */
/*
 * Class:     sample_0005f_tricky_sample_0005f_trickyClass
 * Method:    sample__00024tricky_NativeMethod
 * Signature: ()I
 */
JNIEXPORT jint JNICALL
Java_sample_1_00024tricky_sample_1_00024trickyClass_sample_1_00024tricky
_1NativeMethod
  (JNIEnv *, jobject);
```

The embedded underscores yield different results depending on context. In the case of the:

- file title and class comment, they became "`_0005f`";
- method name comment, they remained "_";
- method signature, they became "`_1`".

Likewise, the embedded dollars yield different results depending on context. In the case of the:

- file title and class comment, they remained "_";

- method name comment, they became "_00024";
- method signature, they became "_00024".

Running `javah sample_$tricky.sample_$trickyClass$sample_$tricky_InnerClass` yields `sample_0005f_tricky_sample_0005f_trickyClass_sample_0005f_tricky_0005fInnerClass.h`:

```
/* Header for class
sample_0005f_tricky_sample_0005f_trickyClass_sample_0005f_tricky_0005fInnerClass */
/*
 * Class:
sample_0005f_tricky_sample_0005f_trickyClass_sample_0005f_tricky_0005fInnerClass
 * Method:    sample__00024tricky_NativeInnerClassMethod
 * Signature: ()F
 */
JNIEXPORT jfloat JNICALL
Java_sample_1_00024tricky_sample_1_00024trickyClass_00024sample_1_00024tricky_1InnerClass_sample_1_00024tricky_1NativeInnerClassMethod
  (JNIEnv *, jobject);
```

No surprises.

Introducing overloaded methods

Another powerful feature of Java is the the possibility of overloading methods. This allows a method (by name) to accept a variety of combinations of input arguments. How does JNI deal with overloaded methods? It appends to the signature a double underscore and a representation of the arguments part of the method signature. Recall that the full signature of a method is "`(<list of argument types>)<return type>`"; only the `<list of argument types>` is appended. Note that you cannot decide whether the overload signature is needed or not unless you check the method in the context of its siblings, something which the design of JavaToDPR 1.0.0 cannot handle.

Type	Signature	Overloaded Method Signature
void	V	
boolean	Z	Z
byte	B	B
char	C	C
short	S	S
int	I	I
long	J	J
float	F	F
double	D	D
<class type>	L<full class name>;	L<full class name>_2
<type>[]	[<type>	_3<type>

What the table does not show is that the package separator (“.” in code) is a slash (“/”) in the signatures, and that the inner class separator (also “.” in code) is a dollar (“$”) in the signatures. Note how the semi-colon translates into “_2” and the opening bracket into “_3”. But how are periods, underscores, and dollars going to be represented in the overloaded method signature?

To see how it goes, we define a test class with overloaded methods:

```
package sample_$tricky;

public class
really_$trickyClass
{
   //Default constructor in use
   native public sample_$trickyClass
   really_$trickyNativeMethod();
   native public sample_$trickyClass
   really_$trickyNativeMethod(sample_$trickyClass[] someObject);

   public class
   really_$trickyInnerClass
   {
      //Default constructor in use
      native public sample_$trickyClass
      really_$trickyNativeInnerClassMethod();
      native public sample_$trickyClass

really_$trickyNativeInnerClassMethod(sample_$trickyClass.sample_$tricky_
InnerClass[] someObject);
   }
}
```

Running `javah sample_$tricky.really_$trickyClass` yields `sample_0005f_tricky_really_0005f_trickyClass.h` (to make the siganture easier to read, I’ve parsed it in alternating colour):

```
/* Header for class sample_0005f_tricky_really_0005f_trickyClass */
/*
 * Class:     sample_0005f_tricky_really_0005f_trickyClass
 * Method:    really__00024trickyNativeMethod
 * Signature: ()Lsample_$tricky/sample_$trickyClass;
 */
JNIEXPORT jobject JNICALL
Java_sample_1_00024tricky_really_1_00024trickyClass_really_1_00024tricky
NativeMethod__
  (JNIEnv *, jobject);

/*
 * Class:     sample_0005f_tricky_really_0005f_trickyClass
 * Method:    really__00024trickyNativeMethod
 * Signature:
([Lsample_$tricky/sample_$trickyClass;)Lsample_$tricky/sample_$trickyCla
ss;
 */
JNIEXPORT jobject JNICALL
Java_sample_1_00024tricky_really_1_00024trickyClass_really_1_00024tricky
NativeMethod___3Lsample_1_00024tricky_sample_1_00024trickyClass_2
  (JNIEnv *, jobject, jobjectArray);
```

The embedded underscores and dollars change according to the rules previously established. The signature appendix is generated from the actual signature (wherein the periods are changed to slashes, whilst underscores and dollars are unchanged) according to these rules:

- underscores become "_1";
- semi-colons become "_2";
- opening brackets become "_3";
- dollars become "_00024";
- forward slashes become "_".

The JNI overloaded method signature consists of:
"Java_<package path>_<class name>_00024<inner class name>_<method name>__<list of arguments signature>".

Running `javah sample_$tricky.really_$trickyClass$really_$trickyInnerClass` yields `sample_0005f_tricky_really_0005f_trickyClass_really_0005f_trickyInnerClass.h`:

```
/* Header for class
sample_0005f_tricky_really_0005f_trickyClass_really_0005f_trickyInnerCla
ss */
/*
 * Class:
sample_0005f_tricky_really_0005f_trickyClass_really_0005f_trickyInnerCla
ss
 * Method:    really__00024trickyNativeInnerClassMethod
 * Signature: ()Lsample_$tricky/sample_$trickyClass;
 */
JNIEXPORT jobject JNICALL
Java_sample_1_00024tricky_really_1_00024trickyClass_00024really_1_00024t
rickyInnerClass_really_1_00024trickyNativeInnerClassMethod__
  (JNIEnv *, jobject);

/*
 * Class:
sample_0005f_tricky_really_0005f_trickyClass_really_0005f_trickyInnerCla
ss
 * Method:    really__00024trickyNativeInnerClassMethod
 * Signature:
([Lsample_$tricky/sample_$trickyClass$sample_$tricky_InnerClass;)Lsample
_$tricky/sample_$trickyClass;
 */
JNIEXPORT jobject JNICALL
Java_sample_1_00024tricky_really_1_00024trickyClass_00024really_1_00024t
rickyInnerClass_really_1_00024trickyNativeInnerClassMethod___3Lsample_1_
00024tricky_sample_1_00024trickyClass_00024sample_1_00024tricky_1InnerCl
ass_2
  (JNIEnv *, jobject, jobjectArray);
```

Invoking JavaToDPR 1.0.3 through `java JavaToDPR -o <dpr name>.dpr -platform Win32 sample_$tricky.really_$trickyClass$really_$trickyInnerClass` correctly generates:

```
library
sample_0005f_tricky_really_0005f_trickyClass_really_0005f_trickyInnerCla
ss;
uses JNI;

(*
 * Class:
sample_0005f_tricky_really_0005f_trickyClass_really_0005f_trickyInnerCla
ss
 * Method:     really__00024trickyNativeInnerClassMethod
 * Signature:  ()Lsample_$tricky/sample_$trickyClass;
*)
function
Java_sample_1_00024tricky_really_1_00024trickyClass_00024really_1_00024t
rickyInnerClass_really_1_00024trickyNativeInnerClassMethod__
  (PEnv: PJNIEnv; Obj: JObject): JObject; stdcall;
begin
end;

 (*
 * Class:
sample_0005f_tricky_really_0005f_trickyClass_really_0005f_trickyInnerCla
ss
 * Method:     really__00024trickyNativeInnerClassMethod
 * Signature:
([Lsample_$tricky/sample_$trickyClass$sample_$tricky_InnerClass;)Lsample
_$tricky/sample_$trickyClass;
*)
function
Java_sample_1_00024tricky_really_1_00024trickyClass_00024really_1_00024t
rickyInnerClass_really_1_00024trickyNativeInnerClassMethod___3Lsample_1_
00024tricky_sample_1_00024trickyClass_00024sample_1_00024tricky_1InnerCl
ass_2
  (PEnv: PJNIEnv; Obj: JObject; Arg1: JObjectArray): JObject; stdcall;
begin
end;

exports

Java_sample_1_00024tricky_really_1_00024trickyClass_00024really_1_00024t
rickyInnerClass_really_1_00024trickyNativeInnerClassMethod__,

Java_sample_1_00024tricky_really_1_00024trickyClass_00024really_1_00024t
rickyInnerClass_really_1_00024trickyNativeInnerClassMethod___3Lsample_1_
00024tricky_sample_1_00024trickyClass_00024sample_1_00024tricky_1InnerCl
ass_2;
end.
```

Overloaded methods sometimes appear in a different order than in the Java source code; I'm not too sure where that comes from. Feel free to "fix" this feature. I'm looking forward to JavaToDPR 1.0.4.

Summary

There are three distinct character transformations going on:

- Header file title (becomes library name) and class comment;
- Method name comment;
- Method signature.

In the first case (header file title and class comment), the rules are (starting from code expression, such as `<package path>.<class name>$<inner class name>`):

- Semi-colons, opening brackets and forward slashes are not possible;
- Underscores become "`_0005f`";
- Dollars become "_";
- Periods become "_".

In the second case (method name comment), the rules are (starting from code expression, such as `<method name>`):

- Periods, semi-colons, opening brackets and forward slashes are not possible;
- Underscores remain "_";
- Dollars become "`_00024`".

In the last case (method signature), the rules are (starting from the signature expression):

- Periods are not possible;
- Underscores become "`_1`";
- Semi-colons become "`_2`";
- Opening brackets become "`_3`";
- Dollars become "`_00024`";
- Forward slashes become "_".

Note that the code expression `<package path>.<class name>$<inner class name>` becomes `<package path>/<class name>$<inner class name>` in a signature.