Technical Note TN2002

Compatibility between JDirect 2 and JDirect 3

CONTENTS

This Technote describes changes in JDirect between MRJ 2.2 on Mac OS 8/9 and the Java runtime on Mac OS X.

JDirect is an Apple technology for calling native code from Java.

All developers interested in creating products in Java using JDirect and making them compatible with Mac OS 8/9 and X will want to review this document.

Updated: [Aug 3 2001]

JDirect 2

JDirect 2 was released with MRJ 2.1. Please see the MRJ 2.2 SDK for more details on JDirect 2.

With JDirect 2 you are required to have a static Object field named `libraryInstance` in each class (or in an interface the class implements) that specifies which shared library(s) to search for native methods. For instance:

```
// JDirect 2 example
import com.apple.mrj.jdirect.JDirectLinker;
import com.apple.jdirect.SharedLibrary;


public class Prime implements SharedLibrary {
  // libraryInstance tells JDirect where to find the
  // native method ComputePrime
  static Object libraryInstance = JDirectLinker.loadLibrary("PrimeLib");

  public static native long ComputePrime(short n);
}
```

Back to top

JDirect 3

JDirect 3 is part of Apple's Java VM on Mac OS X. The main design goal of JDirect 3 was to make it portable to any JNI-compliant VM. It does not require changes to the VM. It is layered on top of JNI.

Note:
Java code written to use JDirect 2 will not run as-is on OS X.

In general, all the features of JDirect 2 are also available in JDirect 3: Native method parameters are marshaled the same, `MethodClosures` are implemented, and the Accessor and Array utility classes are implemented. What is different is the way a class specifies that its native methods use JDirect.

Class initializer

Because JDirect 3 is not integrated into the VM, it does not have a chance to hook up native methods when a class is loaded. Instead, the class needs a static initializer that runs at class load time and tells JDirect to install native method thunks. For example:

```
// JDirect 3 example for Mac OS X
 import com.apple.mrj.jdirect.Linker;

 public class Prime {
    // The call to new Linker() installs the native methods
    static Object linkage = new Linker(Prime.class);

    // This string tells JDirect where to find the native code
    static final String JDirect_MacOSX = "prime.dylib";

    public static native long ComputePrime(short n);
 }
```

The call to `new Linker` registers JNI functions for all native methods in the class. The object returned holds a reference to the memory allocated by the functions installed. By storing the returned Object in a class variable - if the classes are ever garbage collected - the object will be GC'ed. When its finalizer is run, the thunks will be deallocated.

Note:
The client must retain a reference to the returned Object; otherwise, the functions could be free'ed yet still be registered as JNI-native methods!

Library Strings

Because the libraries that implement the native functions have platform-specific names, JDirect 3 uses a system for managing different library names for each platform. JDirect 3 uses reflection to find the platform-specific string in the class with the native methods or in interfaces it implements.

On Mac OS X, JDirect 3 looks for a String named `JDirect_MacOSX`. If JDirect 3 is ported to other VM's, it would look for other specific String names that begin with `JDirect_`.

```
// sample library path
static final String
   JDirect_MacOSX = "/Applications/Foo.dylib";
```

JDirect can be instructed to look in multiple libraries for the native methods of a class by defining multiple `JDirect_MacOSX` Strings each in a different `interface` implemented by the class with native methods. For example:

```
public interface Foo {
    static final String
       JDirect_MacOSX = "/Applications/Foo.dylib";
}

public interface Bar {
    static final String
       JDirect_MacOSX = "/Applications/Bar.dylib";
}

// JDirect will search for these methods in Foo.dylib and Bar.dylib
public class Stuff implements Foo,Bar {
    static Object linkage = new Linker(Stuff.class);
    public native static void NewFoo(int size);
    public native static void NewBar(int size);
}
```

Back to top

Writing code to run on JDirect 2 & 3

It is possible to write Java code that uses JDirect 2 when running on MRJ 2.2 and uses JDirect 3 when running on OS X. The Java VM on Mac OS X contains stubs for all the JDirect 2 classes, but MRJ 2.2 does not contain JDirect 3 classes, which means you can get `ClassNotFoundExceptions` or verifier errors when trying to run code on MRJ 2.2 (which is JDirect 3 aware). The solution is to write your own utility class that uses reflection to find the JDirect 3 Linker class. For example:

```
// utility class that runs on VM's that may or may not have JDirect3
import java.lang.reflect.Constructor;
public class MyJDirectUtil {
  public static Object Link(Class targetClass) {
    try {
      Class linker = Class.forName("com.apple.mrj.jdirect.Linker");
         Constructor constructor
            = linker.getConstructor(new Class[]{Class.class});
      Object jDirectLinkage
            = constructor.newInstance(new Object[]{targetClass});
         return jDirectLinkage;
    }
    catch (Throwable t) {
    }
    return null;
  }
}
```

With the utility class above, you can write classes that mix together JDirect 2 and 3. For example:

```
// interface that works with JDirect 2 & 3
public interface FooLib extends SharedLibrary {
  // for JDirect 2
  static Object
    libraryInstance = JDirectLinker.loadLibrary("FooLib");

  // for JDirect 3
  static final String
    JDirect_MacOSX = "/Applications/Foo.dylib";
}

// class that works with JDirect 2 & 3
public class FooFuncs implements FooLib {
  // links on JDirect 3, but does nothing on JDirect 2
  static Object linkage = MyJDirectUtil.Link(FooFuncs.class);

  public native static void NewFoo(int size);
}
```

Back to top

Carbonized Toolbox

Another big change between MRJ 2.2 and Mac OS X is that the toolbox on OS X is carbonized. That is, only functions and data structures available to Carbon applications are available to Java code via JDirect. For example:

```
import com.apple.mrj.macos.libraries.InterfaceLib;

public class MemFunctions implements InterfaceLib {
   static Object linkage = MyJDirectUtil.Link(MemFunctions.class);

   // will work on MRJ 2.2 and Mac OS X
   public native static int NewHandle(int size);

   // will work on MRJ 2.2
   // will throw exception on Mac OS X because
   //   NewHandleSys is not in Carbon.
   public native static int NewHandleSys(int size);
}
```

Note:
Mac OS X contains a set of compatibilty interfaces in `com.apple.mrj.macos.libraries` which contain a `JDirect_MacOSX` String pointing to the Carbon framework. As in the above sample, this allows code which uses these standard interfaces to work on both Mac OS 8/9 and Mac OS X.

The difference between the toolbox on Mac OS X and Mac OS 9 is not simply that some functions are missing, but some data structures became opaque under Carbon or OS X. If you are using `GrafPorts`, `WindowRecords`, `DialogRecords`, `ControlRecords`, etc., you will need to rework your code to use the new accessor functions (e.g., `GetPortTextFont` instead of `GrafPort.txFont`). With a little more work, you can make your code run on MRJ 2.2 or Mac OS X. Here is an example of how a `Grafport` class could be rewritten to be usable under MRJ 2.2 or Mac OS X:

```
// Option #1: keep GrafPortStruct interface the same
// but change getter/setter methods to use accessor functions
public class GrafPortStruct extends PointerStruct {
   ...
      public final short getTxFont() {
       if (opaqueToolbox)
          return QuickdrawFunctions.GetPortTextFont(this);
       else
             return getShortAt(68);
      }
   ...
}


// Option #2: change all GrafPortStruct clients to use accessors
// style methods which
public class QuickdrawFunctions {
   ...
   public static short GetPortTextFont(GrafPortStruct port) {
      if (opaqueToolbox)
         return GetPortTextFont(port.getPointer());
      else
            return port.getTxFont();
    }

   private static native short GetPortTextFont(int port);
   ...
}
```

Back to top

Method Closures

The class MethodClosureUPP was created for use on Mac OS 8/9 and is not supported on Mac OS X (the design of that class assumes the existence of MixedMode). On Mac OS X, the class exists, but throws an exception if you try to create an instance. If you are using UPP based Toolbox callbacks and want code that will run on Mac OS 9 or Mac OS X, you will need to have two method closure classes - one for JDirect2 that extends MethodClosureUPP and one for JDirect3 that extends MethodClosure. In addition, the MethodClosure for Mac OS X will need to be carbonized which means it uses the appropriate NewXxxUPP and DisposeXxxUPP functions. Please see the sample JDirect code on Mac OS 8/9 and Mac OS X for examples of MethodClosures.

Back to top

Debugging JDirect

The most common problem encountered with JDirect is setting up your classes properly so that JDirect knows which native libraries to search. In both JDirect 2 & 3, JDirect will throw an exception if it cannot find a native method. The exception message will contain a list of the libraries that were searched.

Note:
With JDirect 3, if you forget to call `new Linker` on the class with native methods, you will get the standard `java.lang.UnsatisfiedLinkError` exception that JNI issues, instead of the JDirect exception.

On Mac OS X, you can switch JDirect into verbose mode to aid in debugging. There are two ways to turn on verbose JDirect:

- Set shell variable `JDIRECT_VERBOSE`. This only works if you are starting Java from the command line.

```
> setenv JDIRECT_VERBOSE
> java -cp foo.jar Foo
```

- From your Java code, set `Linker.verbose` to `true`. You will need to open `Console.app` to view output if you do not launch Java from the command line.

```
    com.apple.mrj.jdirect.Linker.verbose = true;
```

When JDirect is in verbose mode, it will write information to `stderr`:

- each time `new Linker` is called, including the libraries that JDirect will search to for native methods in that class and the number of native methods in that class.
- the first time each JDirect native method is called
- each time a `MethodClosure` object is allocated

On Mac OS X, Apple's AWT makes extensive use of JDirect. That means in verbose mode you may see lots of other JDirect usage information besides your own classes.

References

Apple. JDirect Note.pdf.

Apple. Carbon Porting Guide.

Downloadables

Acrobat version of this Note (100K) Download

Back to top
