Software Engineering Concepts (COMP3003)

5. Language Integration

Many large software applications consist of code written in *more than one* language. This is sometimes known as **polyglot programming**[a], and it's done because different languages have different strengths and weaknesses. Using a combination of languages can help you get the best of all worlds.

Here are some situations in which polyglot programming can be useful:

- **Writing plugins, add-ons, extensions, scripts, etc.** These are a commonplace feature in software, small or large. A plugin facility allows third-party software developers to extend your software's functionality without playing with the main code base.

  In principle, the software and its plugins could be written in the same language, but often they are not. In particular, the core code base of a large application is typically written in a statically-typed language: C++, Java, C#, etc. Plugins often use a dynamic language: Python, Javascript, Lua, etc.

  It is hard to nail down scientific evidence for the pros and cons of static and dynamic languages. However, static languages are perhaps better suited for a large, complex code base, because static typing can help avoid certain kinds of type-related defects. Meanwhile, dynamic, interpreted languages are useful for plugins due to their simplicity and accessibility. That is, they can help "lower the bar" for third-party developers to get involved in your projects.

  And there is one more important consideration: plugins should be **sandboxed**, to avoid them becoming a security weakness. Depending on the level of trust you and your users place in third-party developers, you may want to avoid giving them access to command execution, file IO, networking, etc. Sandboxing can be more easily arranged if plugins are written in a different language – you can actually use the language barrier *to your advantage* – although even then it's not something to take for granted.

- **Accessing a library written in one language from another.** In some cases, this does not actually require anything special. You can use C libraries from C++ without writing any C code. You can use Java libraries from Scala or Groovy without writing any Java.

  In other cases, things don't fit together quite as neatly. You cannot generally just use a Java class from C#, or a C++ class from Java, for instance. It's not even the languages themselves that are the problem, technically speaking, but the binary formats they are typically compiled to. But this distinction doesn't really help. In practice, in order to use a library from a different language, you may need to write some "connecting" code in both languages.

- **Balancing high performance requirements with platform independence and memory protection.**

[a] You may come across another, much sillier meaning of the term "polyglot": a source file that is valid in two or more different languages. I leave it as an exercise to the reader to figure out how this can be done, but suffice it to say that it isn't very useful.

If you have in-depth knowledge of optimisation techniques, you may be able to speed-up your code slightly by writing it in a low-level language – C or assembler. However, doing this comes at a cost, and so you would only do it for performance-critical code.

Most of the time, it pays to use a high-level language. Even if your high-level code is marginally slower than it could be, it will be much easier to maintain, much less prone to defects, and typically much easier to port between platforms.

This means that if you *do* use low-level languages for performance-critical code, you will need some cross-language communication.

A special case of this is the use of the graphics processing unit (GPU). GPUs are principally intended for rendering 3D graphics, but they can now be used for general-purpose computation as well. The performance benefits can be significant because GPUs are a whole new computational resource, not just an optimisation. However, since they operate differently from CPUs (performing many calculations in parallel), there are GPU-specific languages, such as "OpenCL C". It may be somewhat difficult to take advantage of GPU computing in, say, Java, C++ or C#.

- **You're writing a software development tool (of some kind).** Dealing with multiple languages could also be a *functional requirement* of your software, if you're writing a compiler, interpreter, IDE, etc. The users may themselves be developers, and may require that your tool deal with several different languages.

  For instance, CPython is a Python interpreter written in C (and is in fact the standard implementation of the Python language). It provides many libraries that are variously written in either C or Python.

1 Native and JVM Code

From the start, Java had two important advantages over C and C++: it was reliably platform independent[b], and it was memory protected.

However, with these advantages came two drawbacks:

- JVM code (usually Java) is slightly slower than native code (usually C/C++). The speed difference is often over-stated. The difference is negligible in many situations, but in computationally expensive algorithms it may become important.

  Memory protection is one of the main culprits. Each time you write "`array[i]`" in Java, the compiler will check that $0 \leq$ `i` $<$ `array.length`, to ensure that you cannot accidentally access memory outside the array. C and C++ compilers have historically not done such bounds checking, and although recent versions *can* do so, this functionality can be turned off if desired.

  (In fact, Java's Just-In-Time (JIT) compiler can actually sometimes skip bounds checking *if* it can prove ahead of time that nothing will go wrong. However, for complex algorithms, such proof may be well beyond the compiler's ability.)

- JVM code cannot simply `import` a native library as it would a JVM one. (You cannot put a native shared library – a .so or .dll file – into Java's "CLASSPATH" and expect that anything will happen.)

So, we need a compromise – a way to call native code from JVM code. In fact, there are two widely-used libraries that allow native↔JVM communication: the Java Native Interface (JNI) and Java Native Access (JNA).

[b] Java achieves platform independence in the time-honoured fashion of *creating a new platform*, but this approach does work.

JNI a standard part of the Java SDK. It works by defining C datatypes and functions that represent JVM objects and operations. To call C functions from Java, those functions must conform to special conventions, and unfortunately (and unavoidably) standard C libraries do not follow these conventions. So, to use these libraries from Java, you must write a small amount of "bridging" C code yourself.

JNA is a 3rd-party Java library. It works by defining Java types and methods that represent C datatypes and functions (more or less the opposite approach from JNI). This allows you to use standard C libraries without writing any C code yourself. This in turn can simplify the build environment. Of course, this only applies if you *don't want* to write C code. If your goal is to write a performance-critical algorithm in C, to be called from Java, then JNA doesn't provide much advantage over JNI. JNA can also incur slightly worse performance than JNI.

(There's a further difference that we shall largely gloss over. The JNI allows native code to actually *start up* the JVM in the first place. So, if you so desired, you could write a C/C++ program that happens to use Java libraries. The JNA does not support this.)

Whichever approach you use, bear in mind that while all platforms should support standard C, they only do so at a source code level. Compiled native code will generally not work on any other operating system, *or* on any other CPU architecture. Even CPUs within the same broad family can have slightly different features. To maximise performance, you might tell your compiler to take advantage of such highly-CPU-specific features. However, this will further limit the range of machines on which your code will run.

Moreover, if you're writing C code yourself (and probably using JNI to call it from Java), you are on your own in C. If you overflow an array, dereference a NULL pointer, etc., you will *not* get a nice Java exception and stack trace explaining what went wrong and where. Yes, you can get a segfault in your Java program this way!

(Advanced C compilers may have their own mechanisms to protect you from such things. However, if you're writing native code for performance reasons, there's a reasonable chance you'll have such options switched off.)

2 Build Considerations

Listing 1 shows how you can arrange the build logic for a project with both Java and C code.

Gradle has several plugins for building native code: "`cpp-application`" (the newest), and also the somewhat older "`c`", "`cpp`", "`assembler`", "`objective-c`" and "`objective-cpp`" plugins. Any of these can be used alongside with the "`java`" plugin. In fact, as far as the build logic is concerned, the C and Java don't really have anything to do with each other.

There are some other noteworthy aspects to Listing 1:

- Gradle will build the C code into a library called "mynativelibrary". (Notice that you specify this name yourself.)
- The C source files will be located in `src/main/c/`, in keeping with convention.
- When using JNI, you must tell the C compiler where to find jni.h, and other files on which jni.h depends. These are included in the standard JDK installation.

Listing 1: A build.gradle file for a project using both Java and C code. (Alternatively, you can use the newer `cpp-application` plugin.)

```
plugins {
    id 'java'
    id 'c'
}

jar {
    manifest {
        attributes 'Main-Class': 'edu.curtin.myapp.Main'
    }
}

model {
    components {
        // Note: 'mynativelibrary' below is a name you pick yourself.
        mynativelibrary(NativeLibrarySpec) {
            sources {
                c {
                    source {
                        // Compile all .c files in src/main/c & subdirectories.
                        srcDirs 'src/main/c'
                        include '**/*.c'
                    }
                }
            }

            // Tell the C compiler where to find 'jni.h' in a system-
            // -independent fashion. (This only applies if you're actually
            // using the JNI, of course, not for C programs in general.)
            binaries.all {
                cCompiler.args "-I$System.env.JAVA_HOME/include"
                cCompiler.args "-I$System.env.JAVA_HOME/include/linux"
                cCompiler.args "-I$System.env.JAVA_HOME/include/win32"
            }
        }
    }
}
```

If you know that the JDK is installed in, for example, "`/usr/java8`", then you could include these lines:

```
binaries.all {
    cCompiler.args "-I/usr/java8/include"
    cCompiler.args "-I/usr/java8/include/linux"
    cCompiler.args "-I/usr/java8/include/win32"
}
```

In Listing 1, we did something slightly more sophisticated, which relies on having the `JAVA_HOME` environment variable set to the JDK location (which it is normally supposed to be).

```
binaries.all {
    cCompiler.args "-I$System.env.JAVA_HOME/include"
    cCompiler.args "-I$System.env.JAVA_HOME/include/linux"
    cCompiler.args "-I$System.env.JAVA_HOME/include/win32"
}
```

(We can do this because (a) Groovy allows us to embed expressions into strings using the `$` sign, and (b) Java/Groovy make environment variables available via the `System` class.)

Gradle will automatically (try to) find an appropriate C compiler; e.g. gcc or clang on Linux, XCode on Mac OS X, or Visual C++ on Windows. There are a few ways to configure the compilation process[c].

3 JNI: Loading Native Libraries

On the Java side, using JNI is reasonably straightforward:

Listing 2: Loading a native library and declaring native methods using JNI.

```
package edu.curtin;
import java.util.List;

public class Attack
{
    // Static initialiser (run when the class is loaded)
    static
    {
        // Load the native code
        System.loadLibrary("mynativelibrary");
    }

    // Declare native methods
    public native static void kapow();
    public native String blam(List<String> list);

    ...
}
```

[c] See the official Gradle documentation for building native code: https://docs.gradle.org/current/userguide/native_software.html.

A `native` method is simply one written in native code. Hence, it has no definition within the .java source file. However, it otherwise behaves like any ordinary method. It can have any parameters or return type, and may be static or non-static. It cannot be abstract (since we're explicitly saying that it *does* have an implementation, just not a Java one), but it may override, or be overridden, or declared `final`.

Let's assume that you've written and compiled a native library called "mynativelibrary". On Windows, the library file would be called "mynativelibrary.*dll*", whereas on Linux it would be "*lib*mynativelibrary.*so*". Calling `System.loadLibrary("mynativelibrary")` will attempt to load that library, and link its functions to the `native` methods declared in Java.

In Listing 2, we put this call in a static initialiser "`static { ... }`" to ensure that the native code is loaded before any methods can be called. (You should put such a static initialiser in each class containing native code.)

Using `System.loadLibrary()`, there will be additional OS-specific setting up required as well. On Linux, you may need to set the `LD_LIBRARY_PATH` environment variable to point to the directory containing "libmynativelibrary.so"[d]:

```
[user@pc]$ export LD_LIBRARY_PATH=build/libs/mynativelibrary/shared
[user@pc]$ java -jar build/libs/myapp.jar
```

As a potentially cleaner solution, you could package up your native library inside your .jar file, have your app extract it to a temporary location, and then run `System.load()`. This (slightly) different function takes a complete filename rather than a library name. It is somewhat beyond the scope of these notes to provide detailed instructions for this.

4 JNI: Native Code Declarations

The C code that implements the native method declarations in Listing 2 would look like this:

Listing 3: Declaring native JNI methods in C.

```c
#include <jni.h>

/* public native static void kapow(); */
JNIEXPORT void JNICALL Java_edu_curtin_Attack_kapow(JNIEnv *env,
                                                    jclass class)
{
    ...
}

/* public native String blam(List<String> list); */
JNIEXPORT jobject JNICALL Java_edu_curtin_Attack_blam(JNIEnv *env,
                                                      jobject this,
                                                      jobject list)
{
    ...
}
```

There is a command-line tool called `javah` that can generate these obtuse signatures from compiled Java code:

[d] This is comparable to Java's `CLASSPATH` environment variable.

```
[user@pc]$ javah -cp build/libs/myapp.jar edu.curtin.Attack
[user@pc]$ cat edu_curtin_Attack.h
```

JNI doesn't actually need the header file that `javah` generates, but it can be useful just to work out what the signatures should be.

So what's going with those JNI function signatures?

- `JNIEXPORT` and `JNICALL` are macros that contain whatever special declarations might be necessary to ensure interoperability with Java. (It is possible that they are actually defined to be empty, but you should still specify them just in case.)
- `Java_edu_curtin_Attack_kapow` (for instance) is the function name, formed as if from the expression `"Java_" + className.replace('.', '_') + "_" + methodName`.

  (C doesn't support function overloading, so if you have overloaded methods, you must append `"__" + mangledSignature` to disambiguate them, but we won't worry about that here.)
- The first parameter is always `JNIEnv*`, called the JNI "interface pointer", which is a starting point for accessing Java objects from C.
- For static methods, the second parameter is of type `jclass`, and contains information (in this case) on the `Attack` class.
- For non-static methods, the second parameter is of type `jobject`, and represents the object instance "`this`".
- The remaining parameters (if any) correspond to the parameters of the Java method.

5 JNI Types

Java and C have very different type systems, and JNI has to map Java's types onto C types[e].

Working with Java's *primitive* types in C is mostly straightforward. In C, JNI defines `jint`, `jlong`, `jshort`, `jbyte`, `jfloat`, `jdouble`, `jboolean`, and `jchar`. Each of these corresponds to a Java primitive type, and *most* can be used just like ordinary C types. Incidentally, `void` has the same meaning in C and Java.

There's also `jsize`, which is really equivalent to `jint` but used to represent sizes, such as the length of strings and arrays.

(Why not just use ordinary C types like `int` and `double`? If JNI used these, things would still generally work fine most of the time in practice. But C imposes much looser standards than Java on the precise definition of `int` and `double`. So, to accurately represent Java types on C, JNI needs to be cautious: `jint` always exactly represents a standard 32-bit Java signed integer, whereas `int` *probably* can in the vast majority of cases, but there's no actual *guarantee*.)

Two types deserve a special mention though:

- `jboolean` is an 8-bit integer which is either 0 (false), or 1 (true). However, for clarity, it's best to use the predefined macros `JNI_FALSE` and `JNI_TRUE`. This adheres to C's assumptions about boolean values (even if C89 doesn't technically have a boolean type).
- Representation of characters is different between C and Java. `jchar` (Java's `char`) is 16 bits wide, and stores Unicode characters in UTF-16 format. C's `char` is only 8 bits wide, and only stores ASCII characters. Standard C functions that expect `char` won't work with

[e] https://docs.oracle.com/javase/10/docs/specs/jni/types.html

`jchar` (or, if they appear to, it will be a fudge that will break horribly on non-ASCII characters).

There is a way of working with Java strings in C, using `GetStringUTFRegion()`, discussed later.

Java's classes, objects, fields and methods are more of a problem, because these don't really correspond to C concepts at all. Instead, JNI provides a series of "handle" types that you use to identify things. You can also think of them as IDs or codes:

- `jfieldID` and `jmethodID` represent particular Java fields and methods. You need these in order to call a method or access a field.
- `jclass` represents a Java class (and corresponds to Java's `Class` class). You need this mostly in order to obtain `jfieldID` or `jmethodID` values.
- `jobject` represents any Java object, whatever its actual Java type.

  JNI also provides a very limited range of more specific object types: `jstring`, `jarray` (and *j*`type`*type*`array`, for various types), and `jthrowable`. In C, these are technically just aliases for `jobject`, and so don't really have any meaning of their own. But using them where appropriate may help to make your code easier to read.

6 JNI Functions

Here's a further example of what you might actually have in one of those C functions from Listing 3:

Listing 4: C code accessing Java objects.

```
JNIEXPORT jobject JNICALL Java_edu_curtin_Attack_blam(JNIEnv *env,
                                                      jobject this,
                                                      jobject list)
{
    // Get reference to the List class
    jclass listCls = (*env)->GetObjectClass(env, list);

    // Get reference to the List.toString() method
    jmethodID toString = (*env)->GetMethodID(
        env, listCls, "toString", "()Ljava/lang/String;");

    // Call list.toString() and return the jobject result.
    return (*env)->CallObjectMethod(env, list, toString);
}
```

```
// The above C code is conceptually equivalent to:
public String blam(List<String> list)
{
    return list.toString();
}
```

It's obvious that the JNI version is extremely verbose, and this is certainly a price you pay for using it. However, the verbosity is only in regards to C-to-Java communication, and doesn't extend to any "normal" C algorithms and operations you might be implementing.

As you can see, JNI's functions are all accessed using the following idiom:

```
(*env)->JNIFunctionName(env, parameters...)
```

As mentioned above, `env` is the JNI "interface pointer", which is needed (a) to *access* JNI functions in the first place, and (b) for those functions themselves for their own internal purposes. In C terms, `env` is a pointer to a pointer to a struct, and that struct contains a large set of pointers to functions, which comprise the JNI API.

In the above example:

- We call `GetObjectClass()` to get the `jclass` handle for the Java's `List`, based on the incoming `list` parameter. *We* know it's a list because that's how the Java method was declared, but the C compiler doesn't know this.

  In fact, technically speaking, it's not `List` itself we're getting here, but a particular implementation of it, such as `ArrayList` or `LinkedList`, depending on how the `list` object was created. But these do, of course, conform to all standard list behaviours.

- Next, we call `GetMethodID()` to get the `methodID` handle for the `toString()` method. We need to supply the class (which we just obtained), the method name, and the method signature or **descriptor**. A method descriptor ("`()Ljava/lang/String;`" in this case) is a compacted representation of the parameters and return type. We'll cover descriptors in more detail shortly.

  It's worth noting that, for *static* methods, you should actually use `GetStaticMethodID()`.

- Finally, we call `CallObjectMethod()` to perform the actual method call. The "`Object`" in "`CallObjectMethod`" refers to the fact that `toString()` returns an object. For a method returning an int, we would instead call `CallIntMethod()`.

  We need to pass the object on which the method is being called – `list` in this case – and of course we must specify the method itself. If there were parameters to `toString()`, we would pass these as well after the method ID.

  (Note: "calling `method()` on object `xyz`" means "`xyz.method()`". The method needs to know its own object, because in most cases it needs to access the object's fields.)

(The JNI calls may seem needlessly complicated. Why the pointers? Why doesn't JNI simply let you write "`jclass listCls = GetObjectClass(list)`"? Basically, flexibility and portability [1, pp. 155]. If we call a function directly in C, the C compiler and linker have to know exactly what that function is. This would mean that our native code would depend, at compile time, on a specific version of the Java virtual machine. That would be a big portability problem, because there are multiple independent JVM implementations out there, and quite possibly even multiple versions on your own computer.)

7 JNI: Method/Field Descriptors

Let's discuss **descriptor**s, which we use to identify methods and fields. They're also used internally in .class files to specify the types of fields, method parameters, and method return types.

A field descriptor is a representation of the field type. Consider the following class fields:

Listing 5: Field descriptor examples.

```
public class Person
{                          // Descriptor:
    private int id;        // I
    private double money; // D
    private String name;  // Ljava/lang/String;
    ...
}
```

We could obtain IDs for these fields as follows:

Listing 6: Getting field IDs.

```
jclass cls = ...;
fieldID idField = (*env)->GetFieldID(env, cls, "id", "I");
fieldID money   = (*env)->GetFieldID(env, cls, "money", "D");
fieldID name    = (*env)->GetFieldID(env, cls, "name", "Ljava/lang/String;");
```

The full mapping of Java types onto descriptors is shown in Table 1. Descriptors *do not* represent generic types, as these are "erased" during the compilation process. Thus, `Set<String>` and `Set<Date>` are both simply "`Ljava/util/Set;`".

Method descriptors involve multiple parts, one for each parameter and the return type, with parentheses around the parameters. The return type can also be "`V`" for void. Here are some examples:

Listing 7: Method descriptor examples.

```
public class MiscClass
{                                                  // Descriptor:
    public void run();                             // ()V
    public double calc(int i, int j, int k) {...} // (III)D
    public char[] read(int i, Reader r) {...}     // (ILjava/io/Reader;)[C
}
```

We could retrieve method IDs for these methods as follows:

Listing 8: Getting method IDs.

```
jclass cls = ...;
jmethodID run  = (*env)->GetMethodID(env, cls, "run", "()V");
jmethodID calc = (*env)->GetMethodID(env, cls, "calc", "(III)D");
jmethodID read = (*env)->GetMethodID(env, cls, "read",
                                     "(ILjava/io/Reader;)[C");
```

8 JNI API

We've already seen a few crucial JNI functions by now `GetObjectClass()`, `GetMethodID()`, `GetFieldID()` and `CallObjectMethod()`. We won't go through all of them (See [f]), but we will now look at a few more crucial ones!

Calling Methods

In general, *non-static* method calls are achieved with a JNI function that looks like this:

[f] https://docs.oracle.com/javase/10/docs/specs/jni/functions.html

Table 1: Java types and descriptors.

Descriptor	Java Type	Notes
`B`	`byte`	
`C`	`char`	
`D`	`double`	
`F`	`float`	
`I`	`int`	
`J`	`long`	
`L`*ClassName*`;`	ClassName	The class must be "fully qualified" (i.e. including its complete package), but with slashes ("`/`") in place of dots; e.g. "`Ljava/util/ArrayList;`" means `ArrayList`.
`S`	`short`	
`V`	`void`	Only valid for method return types.
`Z`	`boolean`	
`[`*Type*	*Type*`[]`	An array; e.g. "`[I`" means `int[]`. The element type can be any non-void type, including a nested array, so that "`[[[Ljava/lang/String;`" means `String[][][]`.

(Adapted from https://docs.oracle.com/javase/10/docs/specs/jni/types.html#type-signatures.)

Listing 9: Calling instance methods.

```
jobject obj = ...;
jmethodID method = ...;
JNIType var = (*env)->CallTypeMethod(env, obj, method, method arguments);
```

JNIType is what we were discussing in Section 5. *Type* is one of `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, `Double` or `Void`. (Although obviously there's no return value in the case of `CallVoidMethod()`.) And *method arguments* are whatever the actual Java method requires.

Static calls are very similar, with a `jclass` instead of a `jobject`:

Listing 10: Calling static methods.

```
jclass cls = ...;
jmethodID method = ...; /* Remember to use GetStaticMethodID(). */
JNIType var = (*env)->CallStaticTypeMethod(env, cls, method, method arguments);
```

There are still more versions of method-calling functions, which we'll very briefly mention. There is an "A"-suffix version of everything previously mentioned; e.g. `CallIntMethodA()`. These accept the *method arguments* in a C array rather than as individual function arguments. There are other versions with a "`V`", in case you'd like to make use of C-style var-args. Then there are the "`CallNonvirtualTypeMethod()`" functions (with "`A`" and "`V`" versions), in case you would like to achieve something similar to Java's "`super.method()`" construct.

Handling Strings

Just as C `char`s are not Java `char`s, C strings are not Java strings.

There are a couple of ways to access Java strings from C. This is (probably) the preferred

approach:

Listing 11: Accessing Java strings.

```
jmethodID = ...;
jstring javaString = ...; /* Likely returned from CallObjectMethod(), etc. */

/* Obtain the number of characters in the string, and the number of bytes
   required to store them: */
jsize nChars = (*env)->GetStringLength(env, javaString);
jsize nBytes = (*env)->GetStringUTFLength(env, javaString);

char cString[nBytes + 1];       /* Allocate a buffer to hold the string. */
memset(cString, 0, nBytes + 1); /* Initialise all chars to NULL. */

/* Get JNI to copy the string into the buffer. */
(*env)->GetStringUTFRegion(env, javaString, 0, nChars, cString);

/* Do important things with cString... */
...
```

There are some notes to make about this:

- Why do we have both `nChars` and `nBytes`? `GetStringUTFRegion()` gives us a "modified UTF-8" string. C can in most cases handle these in a naïve way, by assuming that each byte is a character. And this is true, if your string happens to only contain ASCII (basically English) characters and symbols. However, non-ASCII, non-English characters will take up two or three bytes each. Handling this properly in C is beyond the scope of these notes, but you should be aware of it.
- JNI's string functions *do not* make guarantees about strings having a NULL terminator, as required by standard C functions. This is the reason we allocate an extra byte (`nBytes + 1`) and call `memset()` – to ensure that there will be a NULL terminator.
- The above code allocates a stack-based string, but this may not be wise if your string is very long, and variable-length stack-based arrays are technically illegal in C89 (if adhering to C89 is important to you). So, you could instead allocate your buffer with `malloc()`.

Also, as the name suggests, the same JNI function can give you *substring* too:

Listing 12: Accessing Java *sub*strings.

```
(*env)->GetStringUTFRegion(env, javaString, firstIndex, length, cString);
```

There are several other string functions too: `GetStringChars()`, `GetStringUTFChars()`, etc. These provide similar capabilities in slightly different ways. You may be able to avoid actually copying the string, thus saving some time, but this requires a great deal of care. Consult the JNI specification for more details!

9 Java Native Access (JNA)

JNA[g] is a widely-used 3rd-party alternative to JNI (or, at least, a partial alternative).

Whereas JNI requires the native code to conform to its standards, JNA does not. With JNA, you can use C libraries in Java without writing any C code yourself. Or, if you want to write

[g] https://github.com/java-native-access/jna

the C code, it doesn't have to conform to any particular conventions.

JNA also does not use the `native` keyword. Instead, it requires you to define a Java interface containing declarations for the native functions you need to call. You don't need to write any class that implements that interface, because JNA will autogenerate one on-the-fly, and create an instance of it. The returned object will effectively represent the native code.

JNA is *much* easier to use, but it is not a complete replacement for JNI. In particular:

- In JNA, the interaction is basically one-way. Java can invoke C functions, but the C code cannot access Java classes, objects, fields or methods.
- For ease of use, JNA sacrifices performance. It is noticeably slower than using the JNI.

Therefore, JNA is mostly applicable when you want to use pre-existing C libraries from Java, for their functionality, rather than their performance.

10 Further Reading

I don't have any particular extra references over and above those mentioned in footnotes and in the text. To use any of the libraries mentioned in these notes, you will almost certainly need to consult the relevant online documentation.

10 References

[1] Sheng Liang. *The Java™ Native Interface: Programmer's Guide and Specification*. Addison-Wesley, 1999. **[See section 6.]**