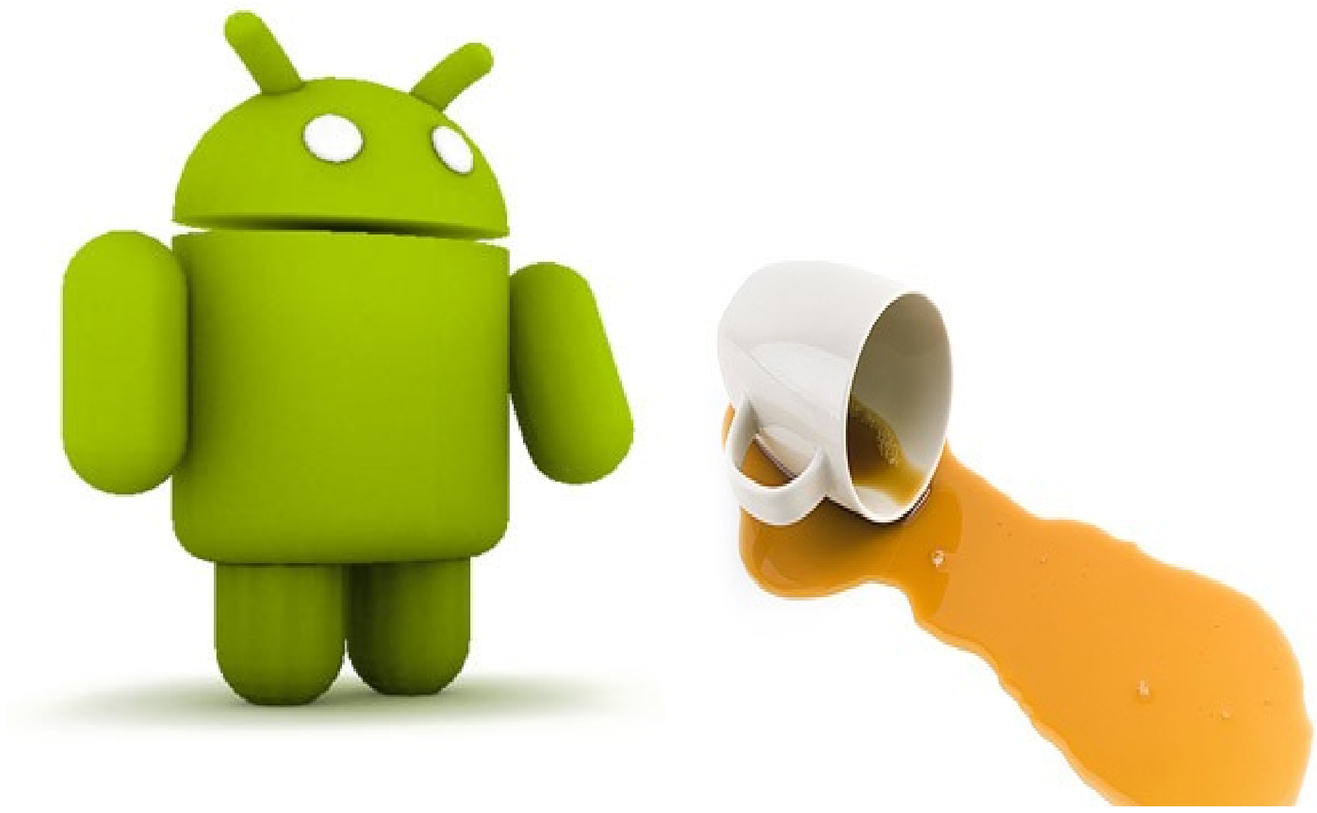
Android without Java



Bernhard "Bero" Rosenkränzer, Linaro
bero@linaro.org

Android Builders Summit 2014

What's this all about?

When starting to write an Android app, the choice of programming languages is clear -- mostly because there's only one option: Java.

...

or are there options after all?

Why would anyone want to use anything else?

Java is a good choice - it is portable across different CPU architectures, virtually all Android documentation explains how to do things in Java, Android APIs were designed with Java concepts in mind, and Java code tends to be readable...

BUT...

Pre-Existing code

Integration with pre-existing code may be necessary/wanted:

- Use of a library not written for Java intended
- Integration with older code may be necessary (e.g. adding a new Android based in-vehicle entertainment system that has to coexist on the same SOC with a pre-existing C/C++ navigation system

Portability

- Do you really need to rewrite that App originally written for that other mobile OS in Objective-C?
- Or the desktop application written in C++ or C?
- Especially if that desktop application will continue to be developed and maintained?

Hardware access

You may be working on a new device with functionalities Android developers never thought of -- and a new kernel (or access library) interface to talk to it: One for which support has never been implemented in Java libraries, and for which no HAL exists so far - so some lower level language code is needed to interface with it.

Special-purpose languages

(Almost) everything can be done in a general purpose language like Java or C/C++ -- but sometimes a special purpose language can be more efficient at a particular task than a general purpose language -- and hand-crafted assembly (or low level C) code can be more efficient than any code generated by a VM.

Habits and personal preferences

If you have a team of C++ programmers, they'll want to use C++ for a new project -- and be much more efficient with it too.

So can it be done?

Yes. In multiple ways.

NDK

To do any of this, you will need an Android NDK -- either Google's official NDK, available at https://developer.android.com/tools/sdk/ndk - or Linaro's NDK, available at http://releases.linaro.org/14.03/components/android/ndk (14.03 is the current release right now - please check for newer releases when trying to download it at a later point!).
There may be other customized versions of the NDK that will do the trick as well.

Modifications in the Linaro NDK: We use Linaro toolchains (with many additional optimizations for ARM architectures), and our gcc config enables support for Objective-C and Objective-C++, to ease porting of applications originally written for that other mobile OS that shall go unnamed. Linaro NDK already contains toolchains based on gcc 4.9.

JNI

JNI (Java Native Interface) is fully supported on Android. It allows calling code with C calling conventions from an application otherwise developed in Java.

JNI in itself is the solution to some of our problems (e.g. using a pre-existing backend library with an otherwise new app)...

It is also at the core of some other solutions (even if they hide it), and there may be a need to call a Java-only API in anything - so let's take a closer look.

JNI: On the Java side

```
package org.linaro.jnitest;
import android.app.Activity;
import android.widget.TextView;
import android.os.Bundle;
public class JNITest extends Activity {
        static {
                System.loadLibrary("JNITest");
        }
        @Override public void onCreate(Bundle s) {
                super.onCreate(s);
                TextView tv = new TextView(this);
                tv.setText(ValueFromCCode());
                setContentView(tv);
        }
        public native String ValueFromCCode();
}
```

JNI - Explanation of Java side

```
static {
        System.loadLibrary("JNITest");
}
```

The `static` block is run once on application startup. `System.loadLibrary` loads a JNI library (which is a regular shared library (*.so) exporting functions that follow JNI naming conventions) into the application, and connects functions exported by the library to Java functions marked "`native`".

```
public native String ValueFromCCode();
```

This is a declaration for a function we'll implement in C -- the "`native`" keyword tells the VM to look for the implementation in libraries loaded with `System.loadLibrary`.

Note that functions are resolved only when called. You don't get a compile time or startup time error if you forgot to write an implementation. (Instead, calling the function will throw an `UnsatisfiedLinkError`).

JNI - C side

```
#include <string.h>
#include <jni.h>
#ifdef __cplusplus
extern "C" {
	jstring
java_org_linaro_jnitest_JNITest_ValueFromCCode(JNIEnv
*env, jobject o);
}
#endif
jstring
Java_org_linaro_jnitest_JNITest_ValueFromCCode(JNIEnv
*env, jobject o) {
	return (*env)->NewStringUTF(env, "This is the
result of a very highly optimized C function");
}
```

JNI - Explanation of C side

The `#ifdef __cplusplus/extern "C"` block makes sure we get C style calling conventions for the JNI functions. JNI doesn't know about C++ calling conventions.
If you're using plain C, you can ignore this part.

```
jstring
Java_org_linaro_jnitest_JNITest_ValueFromCCode(JNIEnv
*env, jobject o) {
```

Various interesting bits in that line:

jstring: this return type is a C interface to a Java `String`

the function name:

```
Java_ org_linaro_jnitest_ JNITest_ ValueFromCCode
```

From Java:

```
package org.linaro.jnitest;
public class JNITest ... {
public native String ValueFromCCode();
```

JNI - Explanation of C code

```
jstring
Java_org_linaro_jnitest_JNITest_ValueFromCCode(JNIEnv
*env, jobject o) {
```

`JNIEnv *env` - we get a pointer to the JNI environment that will do some work for us (such as converting between C types and standard Java types)

`jobject o` - we get a pointer to the object we're part of (o points to "this" -- we don't name it that to avoid conflicts with C++ keywords).

JNI - Explanation of C code

```
return (*env)->NewStringUTF(env, "This is the result of
a very highly optimized C function");
```

This sends our return value -- `NewStringUTF`, passed to us as part of the JNI Environment, converts a C string to a Java String.
In C++, could also use the slightly less ugly

```
return env->NewStringUTF("This is the result of a very
highly optimized C function");
```

In general, for every C callback provided as part of `JNIEnv`, there's a C++ method by the same name that differs only in not having to add the environment as a first argument.
For the rest of this presentation, we'll use the C style calls without pointing out the C++ method (unless it makes a difference).

JNI - Types

Aside from jstring, corresponding to a Java String, you can pass:

Java	**C**	**Java**	**C**	**Java**	C
byte	jbyte	java.lang.Object	jobject	array of longs	jlongArray
short	jshort	java.lang.Class	jclass	array of floats	jfloatArray
int	jint	java.lang.String	jstring	array of doubles	jdoubleArray
long	jlong	java.lang.Throwable	jthrowable	array of chars	jcharArray
float	jfloat	array	jarray	array of booleans	jbooleanArray
double	jdouble	array of ints	jintArray	array of objects	jobjectArray
char	jchar	array of bytes	jbyteArray	indices and sizes	jsize
boolean	jboolean	array of shorts	jshortArray	weak reference	jweak

Always use the correct type -- a jint will likely be the same as an int on your current target platform, but it may not remain the same with a new CPU architecture.

JNI - Type conversion

For integer types, you can use regular C/C++ style casts in JNI code. But Java Strings are rather different from C char* arrays or C++ std::strings.

JNIEnv provides what the functions needed to convert between the two:

```
jstring javaString;
const char *cString = (*env)->GetStringUTFChars(env,
javaString, NULL);
std::string cppString = env-
>GetStringUTFChars(javaString, NULL);
```

Copies the contents of the Java String javaString into the C string cString or the C++ string cppString (internally, this goes through C string conversion, JNI doesn't integrate with the STL directly -- the C function does the same).

```
(*env)->ReleaseStringUTF8Chars(env, javaString,
cString);
```

Tells the VM that we no longer need access to the UTF8 characters (so the string can be garbage collected etc.)

JNI - UTF-8 String conversion

```
jsize s=(*env)->GetStringUTFLength(env, javaString);
```

Gets the length of the string in UTF-8 representation in bytes.

```
(*env)->GetStringUTFRegion(env, javaString, start,
length, buf);
```

Extracts a range of a Java String into the C string *buf* -- *start* and *length* are given as a jsize.

```
jstring s=(*env)->NewStringUTF(env, bytes);
```

Generates a new Java String from the C string in *bytes* (as seen in the example)

JNI - UTF-16 String conversion

```
const jchar *s = (*env)->GetStringChars(env, string,
isCopy);
```

Gets the contents of the Java string referred to by *string* into *s* as an array of UTF-16 characters. *isCopy* is a pointer to a jboolean (may be `NULL`) that will take a value indicating whether *s* contains a copy of the string (*true*) or a pointer to the actual data in the string that can be modified (*false*).

```
(*env)->ReleaseStringChars(env, s, chars);
```

Tells the VM that the native code no longer needs the UTF-16 characters in chars for Java String s. (Same as ReleaseStringUTFChars except UTF-16)

```
(*env)->GetStringLength(env, string);
```

Returns (as a jsize) the length (number of UTF-16 characters) of the Java String *string*

JNI - UTF-16 String conversion

```
(*env)->GetStringRegion(env, str, start, length, buf);
```

Copies *length* UTF-16 characters beginning at offset *start* from the Java String *str* into the C jchar array *buf*. (Similar to GetStringUTFRegion).

```
jchar *utfChars;
jsize length = size;
jstring s=(*env)->NewString(env, utfChars, length);
```

Creates a new Java String from the UTF-16 character array pointed to by *utfChars*, with length *length*.

JNI - UTF-16 String conversion

```
const jchar *cString = (*env)->GetStringCritical(env,
string, isCopy);
```

This is essentially the same as GetStringChars - except you're more likely to get a reference to the data instead of a copy, and you have to be more careful: Between GetStringCritical and ReleaseStringCritical, you cannot cause the thread to block or issue JNI calls.

```
(*env)->ReleaseStringCritical(env, s, cString);
```

The counterpart of GetStringCritical - releases the UTF-16 character array.

JNI - Array conversion

There are identical array conversion functions for all Java primitive types (`int`, `byte`, `short`, `long`, `float`, `double`, `char`, `boolean`) - we'll cover `int` here to demonstrate - all the other types work the same. The array conversion functions are similar to the String handling functions we've looked at before:

```
jint *intArray = (*env)->GetIntArrayElements(env,
array, isCopy);
```

Returns a C style array of `jints` with the contents of Java `int` array (passed as type "`jintArray`" to the JNI function), with isCopy receiving a jboolean indicating whether the C array is a copy or a direct pointer to the data.

JNI - Array conversion 2

```
(*env)->ReleaseIntArrayElements(env, array, intArray,
mode);
```

Tells the VM we no longer need the C array *intArray* referring to the Java array *array*. *mode* is relevant only if the C array is a copy of the Java array, in which case it can take the following values:

0	copy back the content and free the *intArray* buffer
JNI_COMMIT	copy back the content, but do not free the buffer
JNI_ABORT	free the buffer, but don't copy back any changes made.

By default, any code should use 0 because this makes sure the behavior is the same regardless of whether the array was copied or pinned in the first place (something over which the JNI programmer has no control) - the other values can be used for performance tweaking.

JNI - Array conversion 3

```
(*env)->GetIntArrayRegion(env, array, start, length,
buf);
```

Copies *length* elements of the Java int array *array*, starting at offset *start*, into the C int array *buf*.

```
(*env)->SetIntArrayRegion(env, array, start, length,
buf);
```

Copies *length* elements of the C int array *buf*, starting at offset *start*, into the Java int array *array*.

```
jintArray array = (*env)->NewIntArray(env, length);
```

Creates a new Java int array

JNI - Array conversion 4

```
void *cArray = (*env)->GetPrimitiveArrayCritical(env,
array, isCopy);
```

This is essentially the same as `GetIntArrayElements` - except you're more likely to get a reference to the data instead of a copy, and you have to be more careful: Between `GetPrimitiveArrayCritical` and `ReleasePrimitiveArrayCritical`, you cannot cause the thread to block or issue JNI calls.

```
(*env)->ReleasePrimitiveArrayCritical(env, array,
cArray, mode);
```

The counterpart of `GetPrimitiveArrayCritical` - releases the array.

It is noteworthy that, unlike `GetIntArrayElements`, this call is identical for all the primitive types (There's no `GetIntArrayCritical` or `GetFloatArrayCritical`) -- instead, you get a void pointer and have to cast to the correct type.

JNI - Example 2 - Java

```
// package and import statements omitted for brevity
public class JNITest extends Activity {
	static { System.loadLibrary("JNITest"); }
	public native void addOne(int[] numbers);
	@Override public void onCreate(Bundle s) {
		super.onCreate(s);
		int[] numbers = { 1, 2, 3, 5, 7, 11, 13 };
		addOne(numbers);
		StringBuffer sb;
		for(int i: numbers)
			sb.append(Integer.toString(i) + " - 1
is a prime.\n");
		TextView tv = new TextView(this);
		tv.setText(sb.toString());
		setContentView(tv);
	}
}
```

JNI - Example 2 - C

```
// #include statements etc. omitted for brevity
void Java_org_linaro_jnitest_JNITest_addOne(JNIEnv
*env, jobject o, jintArray jArray) {
	jint *cArray = (*env)->GetIntArrayElements(env,
jArray, NULL);
	if(!cArray)
		return NULL;
	for(int i=0; i<(*env)->GetArrayLength(env,
jArray))
		cArray[i]++;
	(*env)->ReleaseIntArrayElements(env, jArray,
cArray, 0);
}
```

JNI - Accessing Java variables

JNI code can access the class it is part of just like equivalent Java code could - but because of missing interoperability, it's not as easy as referencing the member variable by its name or as fast as accessing a C/C++ variable.
You have to go through access functions provided through the JNIEnv pointer:

```
 jclass thisClass = (*env)->GetObjectClass(env, object);
```

Obtains a reference to the object's class to work with (env and `object` are both passed by JNI - they're the first 2 arguments to the function being called).

```
 jfieldID field = (*env)->GetFieldID(env, thisClass,
 "variableName", "fieldDescriptor");
```

Obtains the "field ID" of a member variable - this is a unique identifier that can be passed to accessor functions to reference the class member. Looking up a field ID is a relatively slow operation. You may want to cache the result.

JNI - Accessing Java variables

```
jfieldID field = (*env)->GetFieldID(env, thisClass,
"variableName", "fieldDescriptor");
```

The field descriptor ("filedDescriptor") is the class member's signature (or type) - for a variable, this is a description of its type:

type	**signature**	**type**	**signature**
byte	B	float	F
short	S	double	D
int	I (capital i)	char	C
long	J	boolean	Z

For an object (anything that is not a primitive - even a String), the signature is the letter `L`, followed by the fully qualified name with dots replaced by slashes, followed by a semicolon - e.g. `"Ljava/lang/String;"` for a Java String.
For array types, prefix a `"["` -- e.g. `"[I"` for an array of ints, or `"[Lorg/linaro/test;"` for an array of `org.linaro.test` objects.

JNI - Accessing Java variables

```
jint i = (*env)->GetIntField(env, obj, fieldID);
```

Returns the contents of the class member variable identified by fieldID. As with the array types, this function is available for all primitive types without modification -- GetIntField, GetByteField, GetShortField, ... and GetObjectField for object types.

```
(*env)->SetIntField(env, obj, fieldID, value);
```

Sets the contents of the class member variable identified by fieldID to value - again, this is available for all primitive types and objects (SetByteField, SetDoubleField, ..., SetObjectField)

JNI - static member variables

Trying to use `GetFieldID` and friends on a variable declared `"static"` will result in a nasty surprise - it won't be found.
Static member variables have to be accessed through similar functions:

```
(*env)->GetStaticFieldID(env, thisClass, "name",
"fieldDescriptor");
```

is the same as `GetFieldID`, except it works on static members.

Similarily, to access those members, `Get<Type>Field` and `Set<Type>Field` change to `GetStatic<Type>Field` and `SetStatic<Type>Field`.

An example says more than 10 slides... - Java side

```
 // imports etc. omitted for brevity
public class JNITest extends Activity {
	static {
		System.loadLibrary("JNITest");
	}
	private int i = 42;
	public String s = "Written in Java";
	public static double sd = 3.14;
	protected static String ss = "JNI - Java";


	public native String modifyVariables();
```

Example - Java

```
        @Override public void onCreate(Bundle b) {
                super.onCreate(b);
                TextView tv = new TextView(this);
                String s = modifyVariables();
                s += "After calling C code, variables are:
\n",
                s += "i = " + Integer.toString(i) + "\n";
                s += "s = " + s + "\n";
                s += "sd = " + sd + "\n";
                s += "ss = " + ss + "\n";
                tv.setText(s);
                setContentView(tv);
        }
}
```

Example - C

```
// #includes etc. omitted for brevity
jstring
Java_org_linaro_jnitest_JNITest_modifyVariables(JNIEnv
*env, jobject *o) {
	jclass thisClass = (*env)->GetObjectClass(env,
o);
	jfieldID id_i = (*env)->GetFieldID(env,
thisClass, "i", "I");
	jfieldID id_s = (*env)->GetFieldID(env,
thisClass, "s", "Ljava/lang/String;");
	jfieldID id_sd = (*env)->GetStaticFieldID(env,
thisClass, "sd", "D");
	jfieldID id_ss = (*env)->GetStaticFieldID(env,
thisClass, "ss", "Ljava/lang/String;");
```

Example - C

```
jint c_i = (*env)->GetIntField(env, thisClass, id_i);
jstring c_s = (jstring)(*env)->GetObjectField(env,
thisClass, id_s);
jdouble c_sd = (*env)->GetStaticDoubleField(env,
thisClass, id_sd);
jstring c_ss = (jstring)(*env)-
>GetStaticObjectField(env, thisClass, id_ss);
char *cs_s = (*env)->GetStringUTFChars(env, c_s, NULL);
char *cs_sd = (*env)->GetStringUTFChars(env, c_sd,
NULL);
char buf[512];
snprintf(buf, 512, "Variables initially seen by C code:
\ni=%u\ns=%s\nsd=%f\nss=%s\n\n", c_i, cs_s, c_sd,
cs_ss);
```

Example - C

```
        (*env)->ReleaseStringUTFChars(env, c_s, cs_s);
        (*env)->ReleaseStringUTFChars(env, c_ss, cs_ss);
        (*env)->SetIntField(env, thisClass, id_i, 1701);
        (*env)->SetObjectField(env, thisClass, id_s,
(*env)->NewStringUTF(env, "Written in C"));
        (*env)->SetStaticDoubleField(env, thisClass,
id_sd, 3.1415926);
        (*env)->SetStaticObjectField(env, thisClass,
id_ss, (*env)->NewStringUTF(env, "JNI - C"));
        return (*env)->NewStringUTF(env, buf);
}
```

JNI - Calling Java from C

Another useful feature of JNI is calling Java methods from C - in the Android-without-Java world, this is among the most useful bits, as it allows access to frameworks that provide only Java interfaces.

Accessing methods is very similar to accessing variables:

```
jmethodID mid = (*env)->GetMethodID(env, thisClass,
"name", "signature");
```

and

```
jmethodID mid = (*env)->GetStaticMethodID(env,
thisClass, "name", "signature");
```

get a method ID (just like the field ID for variables)

JNI - Method signatures

Signatures are similar to a variable's signature (field descriptor) as well:
Argument types that need to be passed to the method, enclosed in (parentheses) go first, followed by the return type.
Type encoding is the same as in field descriptors - with an added "V" encoding, indicating a method returning `void` -- e.g.

method	**signature**
`void something()`	`()V`
`String x(int[] i)`	`([I)Ljava/lang/String;`
`org.linaro.TestType[] x(double[] a, String s)`	`([DLjava/lang/String;)[Lorg/linaro/TestType;`
`void x(int a, int b, double c)`	`(IID)V`

JNI - Calling Java from C

Once you have the method ID, you can invoke the method - the syntax is once again similar to getting the contents of a variable:

```
(*env)->Call<returntype>Method(env, obj, methodId,
arg1, ...);
```

where <*returntype*> is the type of the method's return value. It can be one of the primitive types (boolean, byte, char, short, int, long, double, float), object, or void.

Or, for static methods,

```
(*env)->CallStatic<returntype>Method(env, obj,
methodId, arg1, ...);
```

For overridden methods, it is also possible to call the superclass' method rather than the overridden one:

```
(*env)->CallNonvirtual<returntype>Method(env, obj,
methodId, arg1, ...);
```

JNI - Calling Java from C

There are two alternative ways to call a Java method from C code, differing in how arguments are passed:

```
(*env)->Call<returntype>MethodV(env, obj, methodID,
varargs);
(*env)->Call<returntype>MethodA(env, obj, methodID,
jvalues);
```

These functions differ only in how parameters are passed - instead of being passed through "...", Call*MethodV takes a va_list of the arguments, Call*MethodA takes an array of *jvalue*s.

JNI - jvalue

jvalue is a union type supporting all primitives and objects - the naming of union members is similar to the types' signature, but in lower case:

```
typedef union jvalue {
        jboolean z;
        jbyte    b;
        jchar    c;
        jshort   s;
        jint     i;
        jlong    j;
        jfloat   f;
        jdouble  d;
        jobject  l;
} jvalue;
```

Overloading functions

JNI can also provide overloaded functions.
Since C has no concept of overloading, we have to modify the function name. This is done by appending two underscores followed by the function signature. A few modifications to the function signature are necessary to make sure we don't create C syntax errors and create something the JVM can parse:

- A literal _ is replaced with _1
- A ; is replaced with _2
- A [is replaced with _3
- Any non-ASCII character is replaced with _0XXXX, where XXXX is the Unicode character code (hexadecimal using lower case a-f)

Since the return type isn't used in overloading, it (and the parentheses enclosing the parameters) is omitted from the signature.

Exercise: Overloading methods

Try figuring out the correct C function name for the overloaded Java declarations (in package org.linaro.jni, class JNI):

`int x(int a);`	
`String x(float a);`	
`void x(int a, double b);`	
`char x(int[] a);`	
`String x(int a, String[] b);`	
`String x(Test_Class a);`	

Exercise: Overloading methods

Try figuring out the correct C function name for the overloaded Java declarations (in package org.linaro.jni, class JNI):

`int x(int a);`	`Java_org_linaro_jni_JNI_x__I`
`String x(float a);`	
`void x(int a, double b);`	
`char x(int[] a);`	
`String x(int a, String[] b);`	
`String x(Test_Class a);`	

Exercise: Overloading methods

Try figuring out the correct C function name for the overloaded Java declarations (in package org.linaro.jni, class JNI):

`int x(int a);`	`Java_org_linaro_jni_JNI_x__I`
`String x(float a);`	`Java_org_linaro_jni_JNI_x__F`
`void x(int a, double b);`	
`char x(int[] a);`	
`String x(int a, String[] b);`	
`String x(TestClass a);`	

Exercise: Overloading methods

Try figuring out the correct C function name for the overloaded Java declarations (in package org.linaro.jni, class JNI):

`int x(int a);`	`Java_org_linaro_jni_JNI_x__I`
`String x(float a);`	`Java_org_linaro_jni_JNI_x__F`
`void x(int a, double b);`	`Java_org_linaro_jni_JNI_x__ID`
`char x(int[] a);`	
`String x(int a, String[] b);`	
`String x(Test_Class a);`	

Exercise: Overloading methods

Try figuring out the correct C function name for the overloaded Java declarations (in package org.linaro.jni, class JNI):

`int x(int a);`	`Java_org_linaro_jni_JNI_x__I`
`String x(float a);`	`Java_org_linaro_jni_JNI_x__F`
`void x(int a, double b);`	`Java_org_linaro_jni_JNI_x__ID`
`char x(int[] a);`	`Java_org_linaro_jni_JNI_x___3I`
`String x(int a, String[] b);`	
`String x(Test_Class a);`	

Exercise: Overloading methods

Try figuring out the correct C function name for the overloaded Java declarations (in package org.linaro.jni, class JNI):

`int x(int a);`	`Java_org_linaro_jni_JNI_x__I`
`String x(float a);`	`Java_org_linaro_jni_JNI_x__F`
`void x(int a, double b);`	`Java_org_linaro_jni_JNI_x__ID`
`char x(int[] a);`	`Java_org_linaro_jni_JNI_x___3I`
`String x(int a, String[] b);`	`Java_org_linaro_jni_JNI_x__I_3L java_lang_String_2`
`String x(Test_Class a);`	

Exercise: Overloading methods

Try figuring out the correct C function name for the overloaded Java declarations (in package org.linaro.jni, class JNI):

`int x(int a);`	`Java_org_linaro_jni_JNI_x__I`
`String x(float a);`	`Java_org_linaro_jni_JNI_x__F`
`void x(int a, double b);`	`Java_org_linaro_jni_JNI_x__ID`
`char x(int[] a);`	`Java_org_linaro_jni_JNI_x___3I`
`String x(int a, String[] b);`	`Java_org_linaro_jni_JNI_x__I_3L java_lang_String_2`
`String x(Test_Class a);`	`Java_org_linaro_jni_JNI_x__Lorg _linaro_jni_Test_1Class_2`

JNI - Creating objects

It may be necessary to create a Java object from C (or C++, Objective-C, ...) code - JNI allows for this this too:

First, we need to get a reference to the class:

```
jclass cls = (*env)->FindClass(env, "class/name");
```

Here, class/name is the fully qualified class name, with dots replaced with slashes - e.g. if we want a reference to the `java.lang.Integer`, we'd pass "`java/lang/Integer`".

Next, to create an object we'll have to call its constructor (there may be several constructors, so we have to find it through its signature). A constructor is essentially just another method, so we use `GetMethodID` to get the constructor we're looking for. As far as `GetMethodID` is concerned, the constructor is a method called "`<init>`".
Since constructors by definition don't have a return value ("new XYZ()" in Java returns the object, not anything decided on by the constructor), its signature is that of a method returning void.

JNI - Creating objects

```
jmethodID ctor = (*env)->GetMethodID(env, cls,
"<init>", "(I)V");
```

Calling the constructor we've found is similar to calling any other method - but because of the different return type (as with "new", we'll want the class, not any return value), we have to use NewObject (or NewObjectV, NewObjectA) to call the constructor.

```
jobject obj = (*env)->NewObject(env, cls, ctor, ...);
```

where ... is the arguments being passed to the constructor.
It is also possible to create an object bypassing its constructor - but this should usually be avoided (lots of room for error there):

```
jobject obj = (*env)->AllocObject(env, cls);
```

JNI - Creating arrays of objects

The last commonly used type of thing we may want to create from native code is an array of objects. In many ways, this is similar to creating an object.

As before, we'll have to use `FindClass` to get a class reference, and `GetMethodID` to find the constructor we'd like to call.

Then, we'll have to look at 3 new functions:

```
jobjectArray oa=(*env)->NewObjectArray(env, size, cls,
initialElement);
(*env)->setObjectArrayElement(env, oa, index, value);
jobject o = (*env)->getObjectArrayElement(env, oa,
index);
```

`NewObjectArray` creates the array and initializes all members to initialElement (passed as a jobject - typically `NULL`).

JNI - Creating arrays of objects

```
(*env)->setObjectArrayElement(env, oa, index, value);
```

Sets an object inside the array - typically, you'll want value to be the result of a NewObject call, or an object received as input parameter.

```
jobject o = (*env)->setObjectArrayElement(env, oa,
index);
```

Retrieves an object from an object array (unlike the handlers for arrays of primitives, there's no `GetObjectArrayElements` or `GetObjectArrayRegion`).

JNI - References

Now we've done everything that is typically needed for interoperability between Java code and native code -- JNI can do a few more things that allow us to write more efficient and safer code though.

An important thing is handling references - FindClass and friends are expensive operations, so we may want to cache their results. But just caching them in e.g. a global variable in C code would not work because they return a local reference - invalidated once our method exits.

```
jobject something = (*env)->NewGlobalRef(env,
localObject);
```

creates a global reference to localObject that can be cached and will be available until freed with

```
(*env)->DeleteGlobalRef(env, something);
```

JNI - References

JNI also provides a function to delete a local reference (this is done automatically when leaving native code - but you may want to allow a large object to be deleted before allocating lots of additional memory):

```
(*env)->DeleteLocalRef(env, obj);
```

There's also

```
jobject obj = (*env)->NewLocalRef(env, ob);
```

for creating a new local reference from an existing local or global reference. This is useful only in a few special cases (e.g. to make sure a return value is always a local reference that can DeleteLocalRef-ed by other parts of the code).

JNI - local reference lifetime

There are 3 functions to help controlling the lifetime of local references:

```
jint n = (*env)->EnsureLocalCapacity(env, count);
```

Tries to make sure there is space for at least *count* local references (by default, JNI allocates 16 local references upon entry).
Returns 0 on success or a negative number (and an `OutOfMemoryError` exception).

```
jint n = (*env)->PushLocalFrame(env, capacity);
```

Creates a new local reference frame in which at least capacity local references can be allocated. Local references from the surrounding frame remain valid. The return value is the same as that of `EnsureLocalCapacity`. Any local frame created must be closed with:

```
jobject o = (*env)->PopLocalFrame(env, obj);
```

Pops off the current reference frame, freeing all local references. If *obj* is non-`NULL`, returns a reference to obj that remains valid in the new local reference (useful to e.g. keep a result).

JNI - Weak Global References

There is a third type of reference - a so-called weak global reference, generated by

```
jweak w = (*env)->NewWeakGlobalRef(env, obj);
```

and deleted by

```
(*env)->DeleteWeakGlobalRef(env, obj);
```

Like a global reference, a weak global reference is not freed once we return from native code. Unlike a global reference, the garbage collector is free to delete it. C code can check if it has been deleted by using

```
(*env)->IsSameObject(env, w, NULL);
```

to compare the reference to the null object.

JNI - Weak Global References

Caching code could look like this:

```
static jclass cls = NULL;

void Java_org_linaro_jnitest_JNITest_doSomething(JNIEnv
*env, jobject obj) {
        if(!cls || (*env)->IsSameObject(cls, NULL)) {
                jclass c =
                        (*env)->FindClass("org/linaro/Test");
                cls = (*env)->NewWeakGlobalRef(c);
        }
        // Do something fun with org.linaro.Test
}
```

JNI - References

Lastly, we can check what type of reference to something we have:

```
jobjectRefType rt = (*env)->GetObjectRefType(env, o);
```

`rt` can take the values `JNIInvalidRefType`, `JNILocalRefType`, `JNIGlobalRefType` and `JNIWeakGlobalRefType`.

(This is useful e.g. when determining whether we have to use `DeleteLocalRef`, `DeleteGlobalRef` or `DeleteWeakGlobalRef` to make sure the reference can be freed).

JNI - Exceptions

Java relies on exceptions for error handling a lot more than C++, even though the 2 languages' concepts of exceptions are rather similar.

Dealing with exceptions is important if we're interoperating with Java bits - even while we're on the native side.
JNI provides a few functions to handle exceptions:

```
jthrowable ex = (*env)->ExceptionOccurred(env);
```

Checks if an exception occurred. If all is fine, ex will be NULL - otherwise, it contains a reference to the exception: a jobject inheriting java.lang.Throwable. You can use Call*Method etc. on it to find out details.

```
(*env)->ExceptionClear();
```

clears the exception (essentially, this tells the JVM that we dealt with it) and allows the application to go on.

JNI - Exceptions

In addition to catching exceptions thrown by Java code, we can throw our own exceptions back.

```
(*env)->Throw(env, ex);
```

Throws the exception ex, where ex is a completely constructed Java object inheriting java.lang.Throwable.
The easiest use of this is to just pass an exception we got from a Java method call back to Java code invoking us:

```
jobject ex = (*env)->ExceptionOccurred(env);
if(ex)
        (*env)->Throw(env, ex);
```

JNI - Exceptions

We can also create and throw our own exceptions, using

```
(*env)->ThrowNew(env, cls, "Description");
```

where cls is a jclass reference typically obtained through `FindClass`.
A typical use would be

```
jclass exc = (*env)->FindClass(env,
"java/lang/IllegalArgumentException");
(*env)->ThrowNew(env, exc, "Thrown by native code");
```

JNI - Threads

The last important functionality provided through JNI is thread synchronization -- synchronizing on an object -- equivalent to starting and finishing a

```
synchronized(obj) {
        // do something
}
```

block in Java code -- is done through:

```
(*env)->MonitorEnter(env, obj);
// do something
(*env)->MonitorExit(env, obj);
```

Beyond JNI

With all we know about JNI now, it is possible to write large parts of our code in C, C++, Objective-C, Objective-C++ or any other language that supports the creation of shared libraries with C style function names -- but we still need to write some Java code to make use of our functions.

There are ways to move even more functionality to native code.

Native Activity

The NDK includes a header file (platforms/android-*/arch-*/usr/include/android/native_activity.h) that defines structures and functions needed to implement an Activity at the lowest level without writing any Java code. (The Activity contains a global JNIEnv pointer, so we can call Java-only APIs through the JNI functions covered earlier if necessary.)

Outside of the comments in the header file, there is very little documentation on how to make use of this (and there are usually better options).

The main entry point (equivalent of a regular C `main()` function) is `ANativeActivity_onCreate` (name can be changed through the android.app.func_name setting in the manifest), defined with the following prototype:

```
void ANativeActivity_onCreate(ANativeActivity*
activity,  void* savedState, size_t savedStateSize);
```

Native Activity

```
void ANativeActivity_onCreate(ANativeActivity*
activity,  void* savedState, size_t savedStateSize);
```

activity is a pointer to the main native activity object - some members of this struct are pre-filled (so we have access to a JNI environment etc.), others have to be filled by us before doing anything else.

Various callbacks inside the activity should be initialized to functions we're implementing - they're all members of the `ANativeActivityCallbacks` struct accessible through activity->callbacks, so we set our callbacks with

```
activity->callbacks->X = ourImplementationOfX;
```

Native Activity - Callbacks

```
void onStart(ANativeActivity *activity);
```

Is called when the activity has started.

```
void onPause(ANativeActivity *activity);
```

Is called when the activity has paused.

```
void onResume(ANativeActivity *activity);
```

Is called when the activity has resumed from a pause.

Native Activity - Callbacks

```
void *onSaveInstanceState(ANativeActivity *activity,
size_t *outSize);
```

Is called when we should save our current state. We should return a pointer allocated with malloc() [it will automatically be freed by the framework], and set outSize to the number of bytes in the buffer returned.
The state may be saved to disk - so it should not contain anything that will not persist across restarts or reboots, such as pointers or file descriptors.
This data is passed to ANativeActivity_onCreate on the next startup.

```
void onStop(ANativeActivity *activity);
```

Is called when the activity has stopped.

Native Activity - Callbacks

```
void onDestroy(ANativeActivity *activity);
```

Is called when the Activity is being destroyed.

```
void onWindowFocusChanged(ANativeActivity *activity,
int hasFocus);
```

Is called when our window has lost or received focus (used e.g. by games to pause a game when an incoming phone call is interrupting).

```
void onNativeWindowCreated(ANativeActivity *activity,
ANativeWindow *window);
```

Is called when the window for the activity has been created. The information in the "window" struct must be used to draw inside the window.

Native Activity - Callbacks

```
void onNativeWindowResized(ANativeActivity *activity,
ANativeWindow *window);
```

Is called when the window for the activity has been resized (or rotated). The application needs to make sure its rendering matches the new size.

```
void onNativeWindowRedrawNeeded(ANativeActivity
*activity, ANativeWindow *window);
```

Is called when the window needs to be redrawn. In order to avoid transition artifacts, the function should not return until drawing the window in its current state is finished.

Native Activity - Callbacks

```
void *onNativeWindowDestroyed(ANativeActivity
*activity, ANativeWindow *window);
```

Is called when the window for the activity is going to be destroyed. The function must not return before it has made sure nothing (including other threads) will use the window in any way.

```
void *onInputQueueCreated(ANativeActivity *activity,
AInputQueue *queue);
```

Is called when the activity's input queue has been created. The input queue can be used to start retrieving input events.

Native Activity - Callbacks

```
void *onInputQueueDestroyed(ANativeActivity *activity,
AInputQueue *input);
```

Is called when the input queue for the activity is going to be destroyed. The function must not return before it has made sure nothing (including other threads) will use the input queue in any way.

```
void *onContentRectChanged(ANativeActivity *activity,
const ARect *rect);
```

Is called when the rectangle in the window in which content should be placed has changed (e.g. menu buttons or input window taking away space)

Native Activity - Callbacks

```
void *onConfigurationChanged(ANativeActivity
*activity);
```

Is called when the device configuration has changed. New configuration should be retrieved from assetManager.

```
void *onLowMemory(ANativeActivity *activity);
```

Is called when the system is running low on memory - a well-behaved app should release unneeded resources (clear in-memory caches etc.) so the system doesn't have to start killing processes to save memory.

Native Activity - functions

Outside of defining the callback structure, native_activity.h defines a few functions we can call:

```
void ANativeActivity_showSoftInput(ANativeActivity
*activity, uint32_t flags);
void ANativeActivity_hideSoftInput(ANativeActivity
*activity, uint32_t flags);
```

Shows/hides the virtual keyboard. flags can be:

```
ANATIVEACTIVITY_SHOW_SOFT_INPUT_IMPLICIT
ANATIVEACTIVITY_SHOW_SOFT_INPUT_FORCED
ANATIVEACTIVITY_HIDE_SOFT_INPUT_IMPLICIT_ONLY
ANATIVEACTIVITY_HIDE_SOFT_INPUT_NOT_ALWAYS
```

implicit means the action was not manually requested by the user. Forced means it was manually requested.
NOT_ALWAYS means the keyboard remains open if it was FORCED initially.

Native Activity - functions

```
void ANativeActivity_setWindowFormat(ANativeActivity
*activity, int32_t format);
```

Sets the pixel format of the window. There doesn't seem to be a header defining readable constants - but looking at Java code, we can see:

A_8	8	RGBX_8888	2
LA_88	10	RGB_332	11
L_8	9	RGB_565	4
OPAQUE	-1	RGB_888	3
RGBA_4444	7	TRANSLUCENT	-3
RGBA_5551	6	TRANSPARENT	-2
RGBA_8888	1	UNKNOWN	0

Native Activity - functions

```
void ANativeActivity_setWindowFlags(ANativeActivity
*activity, uint32_t addFlags, uint32_t removeFlags);
```

Changes the window flags of the our activity. Available flags:
AWINDOW_FLAG_ALLOW_LOCK_WHILE_SCREEN_ON, _DIM_BEHIND, _BLUR_BEHIND, _NOT_TOUCHABLE, _NOT_TOUCH_MODAL, _TOUCHABLE_WHEN_WAKING, _KEEP_SCREEN_ON, _LAYOUT_IN_SCREEN, _LAYOUT_NO_LIMITS, _FULLSCREEN, _FORCE_NOT_FULLSCREEN, _DITHER, _SECURE, _SCALED, _IGNORE_CHEEK_PRESSES, _LAYOUT_INSET_DECOR, _ALT_FOCUSABLE_IM, _WATCH_OUTSIDE_TOUCH, _SHOW_WHEN_LOCKED, _SHOW_WALLPAPER, _TURN_SCREEN_ON, _DISMISS_KEYGUARD

Native Activity - functions

```
void ANativeActivity_finish(ANativeActivity* activity);
```

Finish (stop and destroy) the activity.

Native Activity - possible pitfall

When using Native Activity directly, callbacks are handled on the application's main thread - if they block, you can get "Application Not Responding" type errors because the main thread will be unresoponsive until the callback has returned.

Android Native App Glue

Android Native App Glue is a small static library that is included in the NDK. It wraps the Native Activity system just discussed in an easier to use way by providing implementations of all the callbacks that send an event to the application's event loop in a separate thread - effectively eliminating the possibility of a callback blocking the main thread causing Application Not Responding errors.

Like Native Activity, Android Native App Glue does not come with a lot of documentation beyond what's in its header file (`sources/android/native_app_glue/android_native_app_glue.h` in the NDK tree), but unlike Native Activity, it comes with example code, located in the `samples/native-activity` directory in the NDK tree.

Let's take a look.

Android Native App Glue

The main entry point for applications using Android Native App Glue is a function called android_main, with this prototype:

```
void android_main(struct android_app* state);
```

The parameter is an "android_app" structure containing references to other important objects, such as the ANativeActivity object we'd get from using Native Activity directly and an ALooper instance (essentially a main loop) that listens to lifecycle and input events.

The ALooper instance can be used to listen for changes in additional file descriptors as well.

android_app

Let's check what's inside the android_app structure android_main receives:

```
void *userData;
```

This is a pointer to any data our app may want to pass around. The Android stack only passes it without doing anything to it - it should typically be used for a global state tracker etc.

```
void (*onAppCmd)(struct android_app *app, int32_t cmd);
```

A callback we can implement to receive application life cycle events. Parameters are the android_app pointer, and the command being run. At this time (Android 4.4 - KitKat, NDK r9d), this is one of:

APP_CMD_* events

APP_CMD_INPUT_CHANGED	The input queue has changed. android_app->inputQueue has the new queue (may be NULL)
APP_CMD_INIT_WINDOW	Our window is ready to use. android_app->window contains the window surface.
APP_CMD_TERM_WINDOW	The current window has to be terminated. android_app->window still contains it, but will be set to NULL after calling android_app_exec_cmd.
APP_CMD_WINDOW_RESIZED	Window has been resized. New size can be found by querying android_app->window.

APP_CMD_* events

APP_CMD_WINDOW_REDRAW_NEEDED	Window needs to be redrawn before calling android_app_exec_cmd()
APP_CMD_CONTENT_RECT_CHANGED	Content area of the window has changed (e.g. input window being shown or hidden). New content rect is in android_app->contentRect
APP_CMD_GAINED_FOCUS	Our app has gained input focus
APP_CMD_LOST_FOCUS	Our app has lost input focus
APP_CMD_CONFIG_CHANGED	Device configuration has changed

APP_CMD_* events

APP_CMD_LOW_MEMORY	We should reduce memory use
APP_CMD_START	Our activity has been started
APP_CMD_RESUME	Our activity has resumed
APP_CMD_SAVE_STATE	We should save our current state (allocated with malloc, will be freed automatically - must not contain pointers, file descriptors etc.) to android_app->savedState and android_app->savedStateSize
APP_CMD_PAUSE	Our activity has been paused
APP_CMD_STOP	Our activity has been stopped
APP_CMD_DESTROY	We're being destroyed. Clean up.

android_app

```
int32_t (*onInputEvent)(struct android_app *app,
AInputEvent *event);
```

A callback we can implement to receive input events. When the callback is invoked, the event has already been pre-dispatched. The callback should return 1 if it handled the event, or 0 to trigger default dispatching (if any).

```
ANativeActivity *activity;
```

The ANativeActivity object from the lower level Native Activity code

```
AConfiguration *config;
```

The device configuration the app is running in

android_app

```
void *savedState;
size_t savedStateSize;
```

Saved state information from the previous instance (may be NULL, e.g. on first invokation or if a previous instance failed to save its state for any reason). The format of this is defined by you - it is whatever you return when getting the APP_CMD_SAVE_STATE event.

```
ALooper *looper;
```

Pointer to the app's main ALooper (event loop) instance.

```
AInputQueue *inputQueue;
```

Pointer to the input queue from which user events are received (may be NULL)

android_app

```
ANativeWindow *window;
```

The window surface our app can draw in (may be NULL)

```
ARect contentRect;
```

The rectangle in which we can place content seen by the user

```
int activityState;
```

The activity's current state. Currently (4.4, NDK r9d), can take APP_CMD_START, APP_CMD_RESUME, APP_CMD_PAUSE or APP_CMD_STOP.

android_app

```
int destroyRequested;
```

Non-zero when the application's NativeActivity is being destroyed.

The android_app structure has a few more members, but those are for internal use by Android Native App Glue only. They're not useful to anything using it, and they may change at any time. Accessing them is not a good idea.

ALooper - the main event loop

After initializing, our Android Native App Glue app needs to listen to the ALooper instance in the android_app object. The main function of interest is

```
int ALooper_pollOnce(int timeoutMillis, int *outFd, int
*outEvents, void **outData);
```

It waits for events to become available, with an optional timeout in milliseconds (a timeout of zero polls the FDs being watched and returns immediately, a negative timeout waits indefinitely until an event appears).

If any event that wasn't handled by a callback occured, outFd, outEvents and outData contain the poll events and data associated with the event.

ALooper_pollOnce return value

ALooper_pollOne returns an identifier >= 0 if its file descriptor has data not already handled by a callback function, or one of

`ALOOPER_POLL_WAKE`	poll was awoken prematurely by something calling `ALooper_wake` on it
`ALOOPER_POLL_CALLBACK`	one or more callbacks were run
`ALOOPER_POLL_TIMEOUT`	the timeout expired without any events
`ALOOPER_POLL_ERROR`	An error occurred.

ALooper

A variant of ALooper_pollOnce is

```
int ALooper_pollAll(int timeoutMillis, int *outFd, int
*outEvents, void **outData);
```

The difference between pollOnce and pollAll is that pollAll doesn't return if a callback has been invoked - it keeps waiting until an event not handled by a callback has occurred, ALooper_wake() is called, or the timeout has expired.

```
void ALooper_wake(ALooper *looper);
```

can be used on any thread to cause the ALooper_poll* calls to return immediately, returning `ALOOPER_POLL_WAKE`.

ALooper

```
int ALooper_addFd(ALooper *looper, int fd, int ident,
int events, ALooper_callbackFunc callback, void *data);
```

adds another file descriptor to be monitored to the event loop. `ident` is an identifier that must be >= 0, and will be the return value of `ALooper_poll*` if an event was triggered. `events` is a bit mask containing events on which we should wake up (`ALOOPER_EVENT_INPUT`, `_OUTPUT`, `_ERROR`, `_HANGUP`, `_INVALID`), `callback` is a callback function to be run on any events (or `NULL`), `data` is custom data passed to the callback.

The prototype for a callback function is

```
int ALooper_callbackFunc(int fd, int events, void
*data);
```

```
int ALooper_removeFd(ALooper *looper, int fd);
```

removes the file descriptor from the ALooper.

Qt

A higher level alternative is using the Qt toolkit (http://qt-project.org/).

Qt is an Open Source (LGPL v2.1 and GPL v3), Cross-Platform UI framework for people using C++ and/or QML (a CSS & JavaScript like language for UI creation). A commercial version is also available.

Qt is very mature and is the base of the KDE, LXQt (also known as LXDE-Qt, the result of merging the LXDE and Razor-Qt) and Hawaii desktop projects.

Recent versions of Qt support building Android apps directly, using the same C++ code you'd run on your desktop device.

Internally, when creating an Android app, Qt creates a small Java shim that takes care of Activity creation and passing events between the Qt event loop and the Android event loop.

Qt

Qt comes with a tool called Ministro that allows the Qt libraries to be shared across several apps using Qt (much like a shared library on a regular Linux system), to avoid bloating every application by having to bundle it with the libraries.

Ministro downloads only the components of Qt actually being used by installed apps.

While Qt is a cross-platform toolkit, it also provides some integration with Android specific features - such as providing access to Android style assets (Assets are treated very much like Qt resources - except they're accessed through "assets:" URLs rather than "qrc:").

Qt is definitely worth checking out, especially for applications that need to run on desktop systems as well.

Apache Cordova

Cordova (http://cordova.apache.org/) is another approach to writing Android apps without Java - it provides a JavaScript API to access device/OS specific functions such as the camera, battery status, accelerometer, files, geolocation, media, and widgets emulating the look and feel of native Android widgets.

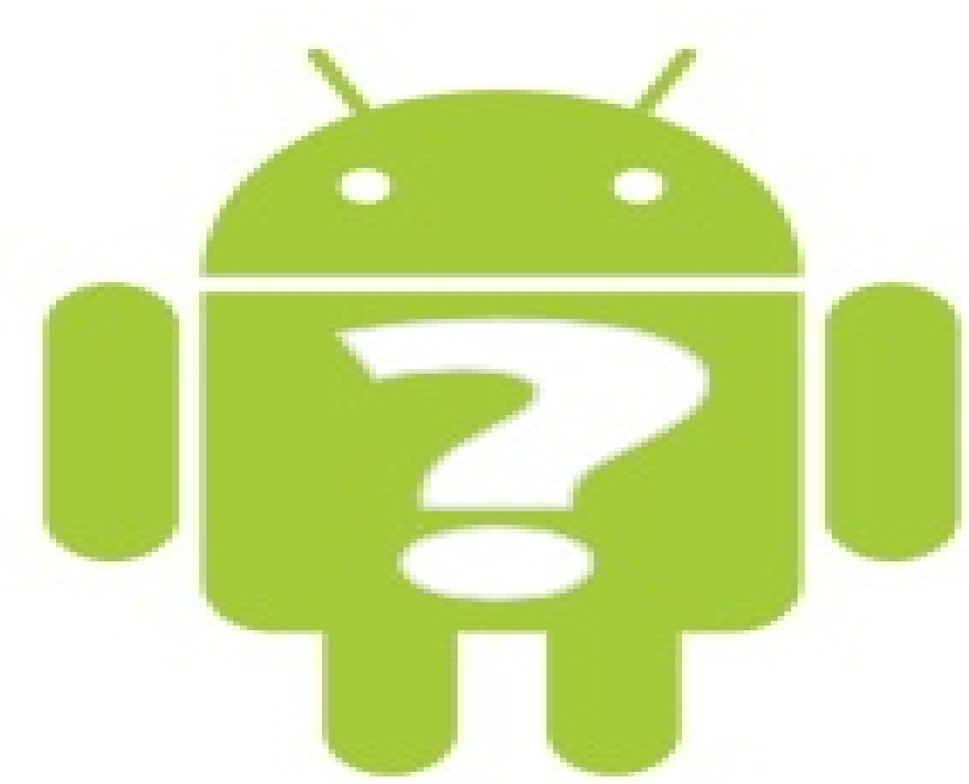
Essentially, this allows writing webapps with extended functionality.

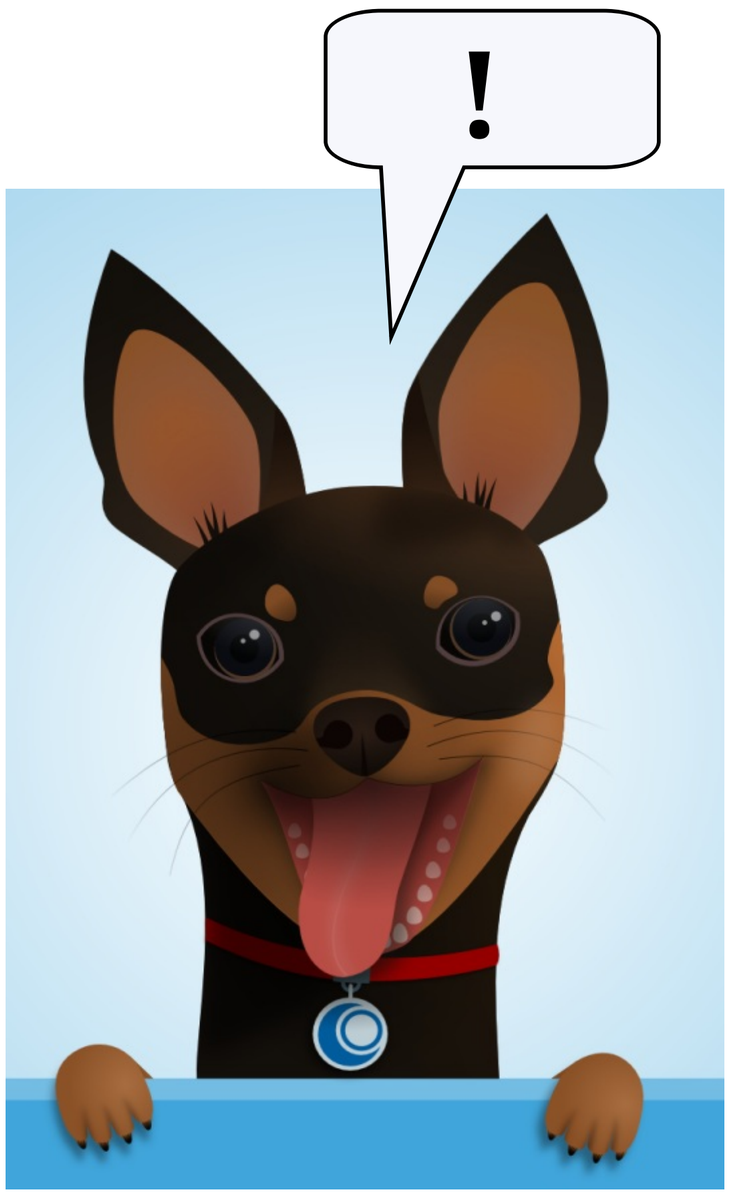
SL4A

The SL4A (Scripting Layer for Android) project, https://code.google.com/p/android-scripting/, aims to bring Python, Perl, JRuby, Lua, BeanShell, JavaScript, Tcl and Shell scripting to Android.

It seems to be in permanent alpha stage and not very active - but is already useful for smaller tasks like learning a scripting language on an Android device.

Questions? Comments?





Bernhard "Bero" Rosenkränzer, Linaro
bero@linaro.org