4.

- Java Native Interface (JNI) is a native programming interface, that allows Java code that runs in a virtual machine to interact with applications and libraries written in other programming languages, like C/C++ and assembler.
- JNI can be used in cases in which the application cannot be fully written in Java.
    - For example, when the Java libraries do not provide support for platform dependant features.
    - Another example is when we use an existing library written in another programming language (it is more memory efficient to load directly a native library than to start a new process in which you load that library).
    - Or when we want to implement a very efficient code written in a low-level language, such as assembler, for performance reasons.
- These are just some examples, there can be other use cases.

5-6.

JNI provides an interface that allows Java applications to invoke native code and vice versa, there is a two-way communication.

1) In this figure, in the first case, we can see how the Java application communicates with the virtual machine that includes JNI, through which it communicates with the native library. Both native and Java code from the virtual machine will get to run on the host machine. The Java application will call native functions in the same way it would call Java methods.
2) In the second case, a native application needs to call a Java library. The native application will be linked to a native library that implements the Java virtual machine and will use the JNI interface to invoke methods from the Java library. For example, a browser written in C needs to execute some applets written in Java by using a native implementation of a Java virtual machine.

7.

When a Java application uses JNI it loses 2 advantages of the Java language:

1) Java applications are portable in the sense that they can run on different platforms for which we have a virtual machine implemented. But the native component will not be able to run directly on another platform. We must recompile the native code for that specific platform.
2) Java language is type safe and secure, but native languages such as C/C++ are not. If a native component makes problems, it can affect the whole application. That is why there are many security verifications that take place when the Java code invokes native code through JNI. We need to be careful when writing the application if we are using JNI.
3) It is recommended to implement native methods in as few classes as possible. This way, we have a clear isolation between the native code and the Java application.

9.

- Android Native Development Kit (NDK) is a set of tools that allow you to embed C/C++ code (native code) into your Android applications.
- You can use it to either build from your own source code or take advantage of the existing pre-built native libraries.
- ndk-build is a shell script that launches the NDK build scripts.
- These scripts will automatically probe your development system and the application project file in order to determine what to build.
- They will generate the binaries.
- And they will copy the binaries into your application's project path.

10.

- This is a minimal example of application that uses the Android NDK.
- First of all, we create the Android.mk configuration file which resides in the jni/ folder.
- It contains two important lines, the one that specifies the native source file (hello-jni.c) and the one that configures the name of the shared library that will be built (hello-jni).
- After building it, the library will have the name libhello-jni.so. The build system will add the lib prefix and the .so suffix.

11.

- Then we create the file Application.mk, which also resides in the jni/ folder.
- This file specifies the CPU and architecture against which to build. In this example, the system will build for all supported architectures.

12.

- The Java source code includes 3 lines that are important.
- First of all, you must have a static initializer in which we load the native library using the method System.loadLibrary() and we specify the name of the library. This code will be executed before any other method.
- Then we must declare the native function. We will use the native keyword which tells the virtual machine that the function implementation is found in a shared native library.
- Finally, we can use the native function directly, in this example will display a message on the screen (It will set the text in a TextView).

13.

And finally we can implement the native source code, which contains the native function.
We observe that we need to include the header file jni.h (which is needed if we want to call JNI functions).
We can see in the example that the native function returns a jstring (which is a JNI data type).
The name of the function is composed of:

- The Java_ string +

- The filepath relative to the top level source directory that contains the Java source code (with underscore instead of slash) +
- the name of the Java source code (without the java extension) +
- the native function name

We have two default arguments (even if the function declared in Java had no arguments):

- JNIEnv * is a pointer to the VM, also called interface pointer
- jobject is a pointer to the implicit “this” object passed from the Java side

The function practically creates a new string using the JNI function NewStringUTF. The function will return this string to the Java code.

15.

- The JNIEnv pointer is the first argument of any native method and is also called interface pointer.
- It is valid only in the current thread that runs the method and will not have any meaning if it is passed to another thread.
- JNIEnv points to a location that includes a pointer to a function table.
- Each entry in this table points to a JNI function, as we can see in the figure.
- The native function can use these JNI functions for accessing or modifying data structures/objects in the Java VM.

16.

- We have a different syntax for calling JNI functions from C and C++ code.
- In C, JNIEnv is a pointer to a JNINativeInterface structure, this is why the pointer must be first dereferenced.
- In addition, JNI functions will not have access to the JNI environment, so we have to give the interface pointer as argument.
- Here, we have an example of calling a JNI function from C. This line of code was included in the previous example.
- In C++, JNIEnv is a C++ class, so the functions are directly accessible as members of the class.
- In addition, JNI functions will have direct access to the JNI environment, so there is no need to give the pointer as argument.
- And here we have an example of calling a JNI function from C++. This is how we are going to represent functions in the next examples.

17.

- The second argument of the native method depends on its type, if it’s static or instance.
- Instance methods are associated with an instance of a class and can be called only on that instance.
- Static methods are not associated with an instance so they can be called directly from a static context.
- Both types of methods can be declared as native and their implementation must be done in the native code.

- The second argument of an instance method is a reference to the object on which the method was invoked, similar with the this pointer in C++. (this is the Java object that includes the native method).
- The second argument of a static method is a reference to the class in which the method was defined (the Java class).
- In the first case it will be a jobject and in the second case it will be a jclass.

19.

- JNI defines a set of C/C++ data types associated to the ones in Java.
- In Java, we have 2 types of data: primitives (int, char, float, etc.) and reference types (classes, instances and arrays).
- The two types of data are treated differently by JNI.
- Mapping the primitives is simple, for example for the int from Java we have jint in C/C++, which is defined as an integer on 32 bits, and the float from Java is mapped on jfloat, which is a floating point number on 32 bits.

20.

- On this slide we have all mappings between the Java primitives and the JNI data types and their meaning. It is interesting that a boolean is represented on unsigned 8 bits and a byte on signed 8 bits, a char on unsigned 16 bits, and a short on signed 16 bits. The rest are pretty intuitive.

21.

- The native methods receive objects in the form of opaque references. These are pointers that refer internal data structures from the Java virtual machine.
- Native code can manipulate these objects only through the JNI functions, that are available through the JNIEnv interface pointer.
- For example, we can use the JNI function GetStringUTFChars() for accessing the contents of a string.
- All JNI references have the type jobject.
- All reference types are subtypes of jobject. These subtypes are associated with the most used reference types in Java.
- For example, jstring is the correspondent of String from Java, and jobjectArray is an array of objects.

22.

- On this slide, we have the mappings between the Java types and the JNI types.
- We observe that only Class, String and Throwable have a specific correspondent.
- Any other object will be identified as jobject.
- We have a large list of object and primitive array types.

24.

- Java strings are handled by JNI as reference type - jstring.
- These strings are not usable directly from the native code as C strings.
- JNI provides the necessary functions for converting Java strings into C strings and vice versa.

- JNI does not provide any method for modifying an existing Java string, because the Java strings are immutable.
- JNI supports both UTF-8 and UTF-16 and will provide functions for handling both types of encodings.
- UTF-16 strings are represented on 16 bits (they are not terminated by \0) and UTF-8 are compatible with 7 bit ASCII (always terminated by \0).
- What is important here is that Java strings (jstring) are represented in the virtual machine in Unicode format.

25.

- A first operation is the conversion of a C string into a Java string, this practically creates a new Java string.
- It is done through the functions NewStringUTF for UTF8 and NewString for UTF-16.
- Here you have an example of how to use this function - we give a C string as argument and it returns a Java string with the type jstring.
- If the virtual machine cannot allocate memory for the new object, it will return NULL and in the virtual machine it will generate an OutOfMemoryError exception.
- So we should always verify if the function has returned NULL and if this happens we should return from the native method.

26.

- In order to convert a Java string into a C string, we will use the JNI functions: GetStringUTFChars for UTF8 and GetStringChars for UTF-16.
- For GetStringUTFChars, we will give the Java string as argument and return a const jbyte * which can be used for displaying the characters with printf if they are only ASCII (7bits).
- GetStringChars returns const jchar * and we can display the characters only with wprintf (UTF-16).
- We also have this isCopy as argument. It will return JNI_TRUE if the string is a copy of the characters from the original instance, and JNI_FALSE if the string is a direct pointer to the original instance (an object pinned in the heap).
- If we are not interested in what it will return, we can pass NULL for this argument.
- If it cannot allocate memory, the call will return NULL and will throw an OutOfMemory exception in the VM.

27.

- After we have finished working with strings, we must free them.
- For this, we need to call the functions ReleaseStringsUTFChars for UTF8 and ReleaseStringsChars for UTF16.
- Here we have an example, it will receive as argument both the Java string and the C string.
- It is recommended to release strings in order to avoid memory leaks.
- If a copy has been made with GetStringChars then it will be freed, otherwise, it will only unpin the object from the heap.

28.

- Let’s see other operations on Strings.
- We can obtain the string length with the functions GetStringUTFLength and GetStringLength on a jstring, or we can call strlen on the result returned by GetStringUTFChars.
- An important operation is to copy parts of the string into a pre-allocated character buffer through the functions GetStringUTFRegion and GetStringRegion.
- Here you have an example, we give as arguments the Java string, a start index, the number of characters that should be copied and a pre-allocated buffer. If we want to copy the whole string, we first must obtain the length.
- The buffer will have the type char [] with a fixed dimension and will be pre-allocated.
- This way, we will not have memory allocations and it is not necessary to make verifications to see if there was enough memory as in the previous slides.

28.

- In order to increase the probability to obtain a direct pointer to an instance of java.lang.String, we can use the calls Get/ReleaseStringCritical.
- They will create a critical region between these calls of functions. In this region, it is not allowed to make blocking operations or wait for other threads from the virtual machine.
- These restrictions make possible the disabling of garbage collection as long as the native code holds a pointer to the string.
- So the native code should not make blocking operations or allocate objects in the virtual machine. Otherwise, you can get a deadlock.
- There is no GetStringUTFCritical because this will most probably make a copy of the string.