4.

- Java arrays are handled by JNI as reference types. We have two types of arrays: primitive and object arrays. They are treated differently by JNI.
- Primitive arrays contain primitive data types such as int, boolean.
- Object arrays include reference data types, such as objects (instances of classes) or other arrays.
- Here you have an example with a vector of objects and a vector of vectors of integers (pay attention that a vector of vectors of primitives is considered a vector of objects).
- We will have different JNI functions to work with the two types of arrays.
- Java arrays are represented through the reference type jarray and subtypes j<primitive>Array and jObjectArray.

5.

- In order to create a new primitive array we have the function New<Type>Array, where type is a primitive data type - int, char, boolean, etc.
- Here we have an example in which we create an array of integers with 10 elements and the returned type of data is jintArray.
- If there is no sufficient memory for allocating the array, the function will return NULL and will throw an exception in the VM. In this case, we must return from the native code.

6.

- JNI provides two methods for accessing the elements from an array. In the first case, we obtain a copy of the data into a C array. In the second case, we obtain a direct pointer to the Java array elements.
- For obtaining a copy, we have the function Get<Type>ArrayRegion.
- Here we have an example, we give as arguments: the Java array, a start index, the number of elements and the native C array. We observe that the buffer is pre-allocated, so the function will not make memory allocations, so we don't need to verify anything.
- After we obtained the C array, we can modify it directly.
- If we want to update the whole Java array, we can use the function Set<Type>ArrayRegion with the same arguments as the previous one. This way, the modified elements can be copied back into the Java array.
- These functions are useful if we have small arrays.
- The longer the array, the more it will take to copy the elements and we will have performance issues. In this case, it is better to work with a direct pointer to the Java array.

7.

- The function Get<Type>ArrayElements will obtain a direct pointer to the array elements when it's possible.
- We have here an example. The function will receive the Java array as argument and will return a pointer to the first element in the array. We will be able to use directly the jint* and work directly on the elements without the need to use a Set function for copying back the elements.

- We observe that the second argument called isCopy has the same meaning as the one from Strings - it will be JNI_TRUE if a copy of the array has been made, and JNI_FALSE if we obtained a direct pointer to the array in the heap.
- So it is not granted that we can obtain a direct pointer to the Java array.
- When we make a copy, it will allocate memory, so we need to verify what the function returns.
- The function returns NULL if there is no memory available.

8.

- An array obtained with Get<Type>ArrayElements must be released using the function Release<Type>ArrayElements.
- Here we have an example. The function will take as parameters both the C array and the Java array.
- The third parameter is the release type.
    - 0 means that we copy back the content and free the native array.
    - JNI_COMMIT means that we copy back the content but we don't free the native array -> just updating the Java array.
    - JNI_ABORT means that we don't copy back the content but we free the native array.
- Other functions for the work with primitive arrays are GetArrayLength that returns the number of elements in the array.
- And Get/ReleasePrimitiveArrayCritical that increases the probability to obtain a direct pointer through disabling the garbage collector. Between these calls we must not make blocking operations or call JNI functions that allocate objects in the VM.

9.

- For working with object arrays, we have the following functions:
- NewObjectArray for creating a new object array. We have as arguments the dimension, the Java class that represents the type of the elements and the initialization value of the elements. It will return a jobjectArray.
- We will not be able to obtain all elements from an array at once, but only one object at a time by using the function GetObjectArrayElement.
- Here you have an example. We give as argument the Java array, and an index and it returns an element, which in this case has the type jstring.
- In C++ we need to make cast to jstring because by default it returns a jobject.
- In the end, we can update an element in the Java array by using the function SetObjectArrayElement. We give as arguments: the Java array, the index and the object that has to be placed on that position.

11.

- Native I/O provides improved performance for handling buffers, scalable I/O operations on files and network.
- NIO buffers provide a better performance than operations on arrays and are recommended when transferring many data between the native code and the Java code.

- The native code can create a byte buffer directly that can be used in the Java application. This buffer will be based on the native byte array.
- Here you have an example. We allocate a native buffer. We remember that a byte in Java is equivalent with an unsigned char in C. The function NewDirectByteBuffer will take as argument this buffer and a dimension, and will return a direct Java buffer.
- The direct buffer can also be created from Java. And the native code can use the function GetDirectBufferAddress for obtaining a pointer to the native byte array. It will receive the Java direct buffer as argument and will return the native byte array.

13.

- In Java, we have two types of fields in classes: instance and static fields.
- In the case of instance fields, each instance will have its own copy of these fields.
- In the case of static fields, all instances will share the the same static fields.
- Here we have examples of defining instance and static fields in Java.
- JNI provides functions for accessing both types of fields.

14.

- First of all, we need to obtain the class of the instance for which we want to modify a field.
- We do this using GetObjectClass. It will receive the instance as parameter and will return an object with the type jclass.
- Then, we obtain an identifier of the instance field by using the function GetFieldID. It will receive as parameters the class, the name of the field and the field descriptor.
- We have Ljava/lang/String; for the String data type. It will return a jfieldID.
- For obtaining the identifier of a static field we can use the function GetStaticFieldID. It receives the same arguments and returns also a jfieldID.
- This ID is used for obtaining the actual field.

15.

- In order to obtain an instance field, we use Get<Type>Field, for example GetObjectField.
- It receives as arguments the instance, the field ID and returns the object of that type, in this example a jstring.
- For obtaining a static field we use GetStatic<Type>Field, for example GetStaticObjectField. It will receive as arguments the class, the field ID and will return an object of this type, in this example jstring.
- In the case in which it cannot allocate memory for the new object, it will return NULL and you must return from the native code.
- Such a call from C to Java will introduce a significant overhead. If we want only to send a value from C to Java, it is better to give it as parameter to the native method.

17.

- In Java, we have two types of methods: instance and static methods. Here we have examples of how to define each method type.
- JNI provides functions for accessing each type of method.

18.

- First of all, we need to obtain the ID of the method.
- For obtaining the ID of an instance method, we can use the function GetMethodID. It will receive as argument the class, the name of the method and the method descriptor, which is some kind of a signature - between the parenthesis we have the argument types and then the return value type. The function will return a jmethodID.
- For obtaining the ID of a static method, we can use the function GetStaticMethodID. It will receive the same arguments and return also a jmethodID.

19.

- For calling an instance method we can use the function Call<Type>Method, in this example CallObjectMethod.
- It will receive as argument the instance, the method ID and will return the result through an object with the type jstring (this is the type of the return value).
- For calling a static method, we have CallStatic<Type>Method, in this example CallStaticObjectMethod. It will receive as arguments the class and the method ID and will return a jstring object.
- In the case when memory cannot be allocated for the new object it will return NULL and you must return from the native code.
- The calls from C to Java also bring significant overhead, and we must not use them often because we can have performance issues.

20.

- On this slide, we have the field and method descriptors, a mapping between the Java type and the signature in C.
- First we have primitives and their associated letter, then a class is represented by L and the class name, array as [and the array type. Methods will be represented as the arguments types in parentheses and then the return type.

22.

- Handling exceptions is an important part of the Java language.
- When an exception is generated, the VM stops the execution of the code block and looks for a block that catches that exception.
- In that moment, the VM erases the exception and gives control to the block that handles the exception.
- In contrast, on the native side, the developer needs to explicitly implement this process when a (Java) exception is generated, for example when calling a Java method from C.
- We will be able to catch an exception generated when calling a Java method by using the function ExceptionOccurred.
- Here we have an example, we call the method and then verify if an exception has occurred. If so, it will delete the exception and execute the appropriate code for the case.

23.

- We can also throw Java exceptions in the native code.

- But first, we have to obtain the class of the exception through the function FindClass.
- Then we throw the exception by using the function ThrowNew. The function will have as arguments the exception class and the message that is sent with the exception.
- Throwing an exception will not automatically stop the native code and will not give control to the exception handling block. Therefore, after throwing an exception, we must release the resources and call return in order to terminate the native method.

25.

- References have an important role in Java. The VM handles the lifetime of an instance by monitoring the reference and performing garbage collection when it is not referred anymore.
- JNI provides functions for handling references and their lifetime.
- JNI supports 3 types of references: local, global and weak global.

26.

- Most JNI functions return local references. They cannot be reused in further invocations of the native method, because their lifetime is limited to that method and they will be released when the method terminates.
- Theoretically, you can use minimum 16 local references in the native code, but it is recommended that when using many references in operations that work intensively with the memory, to free the local references when possible.
- For manually releasing a local reference, you can use the function DeleteLocalRef.
- Here you have an example. We obtain a local reference using the function FindClass and then we free it. The function will receive the reference as argument.

27.

- Global references remain valid in the further invocations of the native method until they are explicitly freed.
- A global reference must be created with the function NewGlobalRef.
- Here you have an example. We obtain a local reference through the function FindClass (or any other function that returns a reference) and then we call NewGloabalRef that will take the local reference as argument and will return a global reference.
- In the end, we free the global reference.
- When we finish using a global reference, we should free it through the function DeleteGlobalRef that will receive the global reference as argument.
- Global references can be used also by other methods, or from other threads, as they are not limited to the method in which they were created.

28.

- A weak global reference is valid in the further invocations of the native method, but the referred object can be garbage collected at any time.
- For creating a weak global reference, we can use the function NewWeakGlobalRef that receives as argument a local reference.

29.

- In order to verify if a weak global reference is still pointing to an instance, we can use the function IsSameObject.
- In this example, we compare the reference with NULL using the function IsSameObject. If the reference is not equal to NULL, it means that the instance is still alive and can be used.
- If it is equal to NULL, it means that the instance has been garbage collected and cannot be used anymore.
- For deleting a weak global reference we can use the function DeleteWeakGlobalRef that receives the reference as argument.

31.

- The native code can be multithreaded. However, we have the following constraints regarding references and JNIEnv.
    - Local references are valid only in the native method and on the current thread that executes the method.
    - Local references cannot be shared between multiple threads.
    - In contrast, global references can be shared between threads.
- The interface pointer JNIEnv is given as argument to the native method and is valid only in the thread associated with that native method. It cannot be used by other threads.

32.

- The Android applications can use threads for executing tasks in parallel.
- All threads (Java, native) are Linux threads scheduled by the Linux kernel.
- Threads can be started from the Java code using Thread.start or from the native code using pthread_create.
- When a thread is created from the native code, the VM will not know about it until it is attached to the VM. Until then, it does not have access to the interface pointer JNIEnv and it cannot call JNI functions.

33.

- For attaching a thread to the virtual machine we can use the function AttachCurrentThread or AttachCurrentThreadAsDaemon.
- Here we have an example of attaching a thread to the VM. The function is called on the JavaVM pointer This way, we can obtain an interface pointer JNIEnv valid for the current thread and we will be able to call JNI functions.
- This operation creates a java.lang.Thread object in the VM and attaches it to the main ThreadGroup. We can use the last argument, which is a JavaVMAttachArgs structure, to specify other thread group
- Then, we can perform operations on that thread, and in the end, we must detach the thread from the VM by using the function DetachCurrentThread of the JavaVM pointer.

34.

- Synchronization is very important in multi-threading environment. We will be able to synchronize the native code using monitors that are based on Java objects.

- A single thread will be able to hold the monitor at a certain moment in time, and the other threads that want to obtain the monitor will wait until it is released.
- For obtaining a monitor we can use the function MonitorEnter that receives as argument a Java object.
- If another thread holds the monitor, the thread will wait until it is released. If no other thread holds the monitor, the current thread will obtain it and set the entry counter to 1. If the current thread already holds the monitor, then the entry counter will be increased.

35.

- For releasing a monitor, we can use the function MonitorExit, which receives as argument the same Java object.
- For this operation to succeed, the current thread must hold the monitor.
- In this call, the entry counter is decremented and if it reaches zero, the current thread releases the monitor completely.

37.

- All options from JNI 1.6 are implemented in Android, excepting DefineClass through which JNI loads a class into a buffer. In Android, we do not work directly with bytecode or with class files. Therefore, this function is useless in Android.
- JNI makes very few error verifications. Usually, when an error occurs, the application crashes.
- Android provides a mode called CheckJNI that performs a series of verifications before calling JNI functions.
- CheckJNI is enabled by default in the emulator and can be enabled on physical devices from the command shell.

38.

We will enumerate some of the verifications performed by CheckJNI:

- It will verify if there is an attempt to allocate arrays with negative size.
- if there is an attempt to pass an invalid pointer to a JNI function
- passing a NULL argument to a JNI function when the argument must not be NULL
- passing an invalid class name to a JNI function
- calling a JNI function in a critical region
- passing invalid arguments to NewDirectByteBuffer
- calling a JNI function when an exception has been generated (is pending)

39.

- when using JNIEnv on another thread
- when using a NULL, an invalid field ID, or static instead of instance and vice versa
- when using an invalid method ID, having an incorrect return type, using static instead of instance and vice versa, or having an invalid class or instance
- when calling DeleteGlobalRef or DeleteLocalRef on the wrong type or reference
- when passing a bad release mode
- when returning a value with the wrong type from a native method
- when giving an invalid UTF-8 string to a JNI function