4.

- Starting from Android 2.3 Gingerbread, (API level 9), native activities have been introduced. They allow the implementation of Android applications only in native code, without a single line of Java code, only C/C++.
- NDK provides an API for accessing Android resources, like displaying windows, accessing assets, configuring the device.
- However, many functionalities are missing and Java remains the main language for developing the graphical interface.
- Native activities can still be used for developing multimedia applications.

6.

- First, we will see how to create low-level native activities using an example.
- We have an application named NativeApp (any name here). We created the project without any Java activity. We right click on the project, Android Tools -> Add Native Support, which will add the jni folder with a source file and Android.mk.
- In the Manifest file, we need to specify the minimum API level 9. The name of the activity needs to be exactly android.app.NativeActivity. We need to specify the property android.app.lib_name to be exactly the name of the native module without the lib prefix and the .so suffix, for example NativeApp (like the name of the application). Also hasCode needs to be false (because we don't have any Java code).

7.

- Here we have an implementation example. We notice that in ANativeActivity_onCreate we will associate a series of callbacks that will be implemented in this source file.
- We observe the implementation of the onStart callback. In each callback we implement the behavior of the application when the associated activity lifecycle event takes place.

8.

- We will have this source file NativeApp.cpp in the jni/ folder.
- We saw the implementation of the function ANativeActivity_onCreate, which is the entry point in the activity.
- The function has 3 arguments:
  - ANativeActivity structure which is defined in the native_activity.h (which has to be included in the source file).
  - savedState which is the previous saved state of the activity
  - savedStateSize which is the dimension of the previous state (in bytes)

In this function, we will associate callbacks that will handle the lifecycle events and the user's input.

9.

- The Android.mk file will look like this, nothing out of the ordinary.
- Then, we can compile and run. And here you have part of the output.

- The first three lines are generated by the function printInfo, that will display the fields from ANativeActivity.
- The next 3 lines are displayed at Start and Resume (that is the address of the activity).

10.

- The Android framework includes the class android.app.NativeActivity, which is a subclass of android.app.Activity.
- This class helps create the native activity, more exactly, it is a wrapper that hides the Java world from the native code and exposes native interfaces defined in native_activity.h.
- When we want to start the native activity, an instance of that class is created, which calls ANativeActivity_onCreate through JNI.
- The NativeActivity instance will also invoke callbacks for treating the events that take place.

11-12.

This is how ANativeActivity structure looks like (defined in native_activity.h):

- We have a vector of pointers to callback functions. We will set these pointers to our callback functions. The callbacks will be called by the NativeActivity instance (from the Android framework).
- vm is the interface pointer JavaVM, which is a handle to the virtual machine
- env is the interface pointer JNIEnv, which can be used for calling JNI functions
- clazz is a reference to the NativeActivity object, and can be used for accessing methods and fields in the instance
- internal/externalDataPath
- the SDK version of the device
- instance - a pointer that can be used for storing application specific data
- assetManager - can be used for accessing binary assets from the assets folder in the apk.

14.

- Now moving on to the high-level native applications.
- So far, we talked about low-level native applications in which we manually assigned callbacks for the events.
- This mechanism, defined in native_activity.h, uses the main thread for the lifecycle events and for handling the user's input.
- If we have long callback functions, the application will not answer to the user's input anymore.
- The solution is to use multiple threads.
- This is done by the mechanism implemented in the android_native_app_glue library, which is built on top of native_activity.h
- It will use a separate thread in order to execute callbacks and handle the user's input.

15.

- Here we have an example.
- In high-level native applications, the entry point in the application is android_main, which receives an android_app structure.
- First of all, it calls app_dummy(), which is a function from the API defined in android_native_app_glue.h, that does nothing at all. This call must be present in order to verify if the Android build system includes android_native_app_glue.o (if glue is stripped).
- We put in userData a pointer to an integer and in onAppCmd a handle of activity lifecycle events.
- In a loop, we will interrogate the looper if an event has been received. If so, then it runs the appropriate handler, in this case handle_activity_lifecycle_events, which will display the code of the command and the data saved in userData.

16.

- First, let's see some implementation details for high-level native applications.
- First of all, we need to implement android_main, which includes a loop in which we will interrogate the looper for the events that were received.
- android_main will run on the background thread.
- By default, we have two event queues attached to the background thread. A queue for the lifecycle events and a queue for the user input events.
- We can determine the type of event through an identifier that can be LOOPER_ID_MAIN or LOOPER_ID_INPUT
- In addition, we can have other event queues if they are necessary in the application.
- The field userData is used for transmitting data to the event processing function (a pointer).

17.

- When an event is received, we will get a pointer to a structure.
- In this moment, we will call the processing function, called process, that will point to android_app->onAppCmd for activity lifecycle events and to android_app->onInputEvent for user input events.
- We will have to implement our own processing functions and set the pointers to them.
- In our example, we implement the function handle_activity_lifecycle_events and we set onAppCmd to point to it, and this means that it will handle the activity lifecycle events.
- Our function will display the cmd value and the saved data.
- Cmd is defined in android_native_app_glue.h and signifies a certain event.
- For example APP_CMD_START = 10, APP_CMD_RESUME = 11, etc. This is how we can determine the event that took place.

18.

- The Android.mk file is similar, we need only to specify that the application is linked to the static library android_native_app_glue.

- If we compile and run, we will observe the following output: the first two are the start and resume events, the next two are for window initialization and obtain focus.

19.

- Let's see some details about the android_native_app_glue.
- It implements the function ANativeActivity_onCreate, implements and registers callbacks, and then calls the function android_app_create.
- The function android_app_create initializes a structure android_app which is defined in android_native_app_glue.h, then creates a unidirectional pipe for the communication between threads.
- Finally, it creates a new thread, called the background thread, in which the function android_app_entry will run, and will receive the structure android_app as argument.
- The pipe is used for communicating between the main thread (that runs android_app_create) and the background thread.
- The function android_app_entry will create a looper and will attach the two event queues to this looper. Finally, it will call the function android_main that includes our implementation.

21-22.

- Next, we will talk about handling windows in native applications.
- Here we have an example:
- In the handler for the lifecycle event, if cmd is APP_CMD_INIT_WINDOW, then we call a function that will draw a colored rectangle.
- First, we obtain the window associated with the activity, then we set the window dimension and its format: 0,0 means that width and height are the ones of the screen, and then we choose one of the 3 formats (RGBA8888 - 8b R, 8b G, 8b B, 8b A; RGB565 - 5b R, 6b G, 5b B; RGBX - last 8b ignored).
- Then we lock the window for drawing and we get the buffer of the window.
- We initialize the buffer, we set the width and height of the rectangle and it's location on the screen.
- Then we configure the color of each pixel by setting a value for red, green, blue and alpha (transparency).
- Finally, we unlock the window and draw it.

23.

- And this is a screenshot of the resulted application.

24.

- Let's see all steps in detail.
- First of all, we need to include the header file native_window.h, which defines the functions for working with windows.

- We will use the function ANativeWindow_setBuffersGeometry for setting the window format and its dimension using the arguments: ANativeWindow type, width and height, the format which can be one of the 3 options.
- Then we will lock the next drawing surface using the function ANativeWindow_lock that will return as argument a window buffer with the type ANativeWindow_Buffer.

25.

- Then, we need to clear the buffer. We can erase the whole buffer or just a part from it. To erase the whole buffer we can initialize it with zero using memset.
- Then, we will draw in the buffer by:
    - Configuring the width and height of the rectangle
    - computing the start and the end for the width and height - the 4 corners of the rectangle
    - Then for each pixel in the rectangle, we will set the bytes for red, green, blue and alpha.
- Finally, we will unlock the surface and draw the buffer on the screen.

27-28.

- Next, we will talk about handling the user's input.
- Here we have an example. In android_main, we will set onInputEvent to point to the function handle_input_event.
- In this function, we will first obtain the type of the event that can be either a key event or a motion event.
- If it is a motion event, we will obtain the action (which is in fact the action + a pointer index), and we will extract the pointer index based on the action and some predefined flags (we have the actual action on 8 bits and the pointer index on the next 8 bits).
- Then, we will obtain the value of x. If the action was one in which the finger was moved on the screen, then it computes the new position of the object on the screen.
- In android_main will be called a function that re-draws the object on the screen on the new position.

29.

- Now let's see some implementation details.
- First of all we need to set a handle for the user's input events, so the onInputEvent will point to our handle function.
- In the handle function, we obtain the event type using the function AInputEvent_getType. The type can have one of two options: key event and motion event. The first appears when the user presses a key (on a physical or software keyboard) and the second when the user touches or moves the finger on the screen.
- We can obtain the identifier of the device that generated that input (keyboard, touchscreen, mouse, touchpad) through the function AInputEvent_getDeviceId.
- Next we will see the functions that can used for obtaining more information.

30.

- Let’s see a part of the API for key events.
- For obtaining the action - AKeyEvent_getAction - which can be up, down or multiple (multiple presses).
- For obtaining the flags - AKeyEvent_getFlags - the flags give us additional information, for example: if a software keyboard was used, if the events were generated by the system, if it was a longpress, etc.
- For obtaining the code of the pressed key - AKeyEvent_getKeyCode - each key has a code associated and we can identify exactly which key was pressed.
- We can see how many times the event was repeated - AKeyEvent_getRepeatCount - the number of up, down and multiple presses.
- We also can obtain the time when the event appeared - AKeyEvent_getEventTime.

31.

- Now let’s see a part of the API for motion events.
- With AMotionEvent_getAction, we can obtain the action combined with the pointer index. The action can be down, up, none, cancel, etc. From the returned value we can extract the pointer index exactly like in the example.
- We can obtain the flags with AMotionEvent_getFlags.
- Important - we can obtain the value of the x coordinate for a certain pointer index, by using AMotionEvent_getX. A positive value represents a pixel, and on certain devices we will obtain a fraction which represent subpixels.
- In the same way we can obtain the Y coordinate with AMotionEvent_getY.
- For obtaining the pressure of the screen touch, we use AMotionEvent_getPressure.

33.

- Finally, we will talk about handling assets.
- We have here an example, in which we read a text file that is found in the assets/ directory in the apk.
- First of all, we need to obtain the AssetManager - we have a pointer to it in a field of ANativeActivity.
- Then we open the assets/ directory using AAssetManager_openDir.
- We obtain the name of the first file in that directory with AAssetDir_getNextFileName.
- Then, we open the file using AAssetManager_open
- We obtain the dimension of the file using AAsset_getLength.
- We allocate a buffer with that dimension.
- We read the whole file using AAsset_read.
- Finally, we close the file using AAsset_close and the directory using AAssetDir_close.

34.

- We can use this API for accessing the files in the assets/ directory (text, audio, video and images). Let’s see the API in more detail.

- First of all, we need to obtain a pointer to the AAssetManager. From a native library, we can obtain it from Java through the JNI function AAssetManager_fromJava. From a native application, we obtain it from the field of the activity->assetManager.
- We can open the assets/ directory or a subdirectory through the function AAssetManager_openDir. If we give as argument “” -> the assets/ directory, otherwise we specify a subdirectory from assets/.
- For iterating through the files of an open directory we can use AAssetDir_getNextFileName, that will return a filename. It will return NULL when we went through all files and there is no other file in the directory.

35.

- In order to open a file, we will use AAssetManager_open. We need to specify the access mode to that file:
    - unknown - if we don’t know how we are going to read the file
    - random - if we read pieces, move forward and backward
    - streaming - if we read sequentially, and move only forward
    - buffer - we read the whole file in the memory - this is for quickly loading small files.
- For reading from a file we can use AAsset_read. The syntax is similar to read, it reads into a buffer.
- In the end, we close the file with AAsset_close and the directory with AAssetDir_close.
- It is important to know that we cannot write into a file from assets.