

Mobile Computing
Professor Pushpendra Singh
Indraprastha Institute of Information Technology Delhi
Lecture 28
Saving Data

Hello, today we will start a completely new topic but a very important one. Most of your android applications require some data to be saved. So far we have been working on very simple application which did not have these requirements. However, now we will be developing applications which will require some data to be saved. Android provides us multiple ways to save data and retrieve it and in this lecture we will learn about it, so let us get started.

(Refer Slide Time: 0:57)

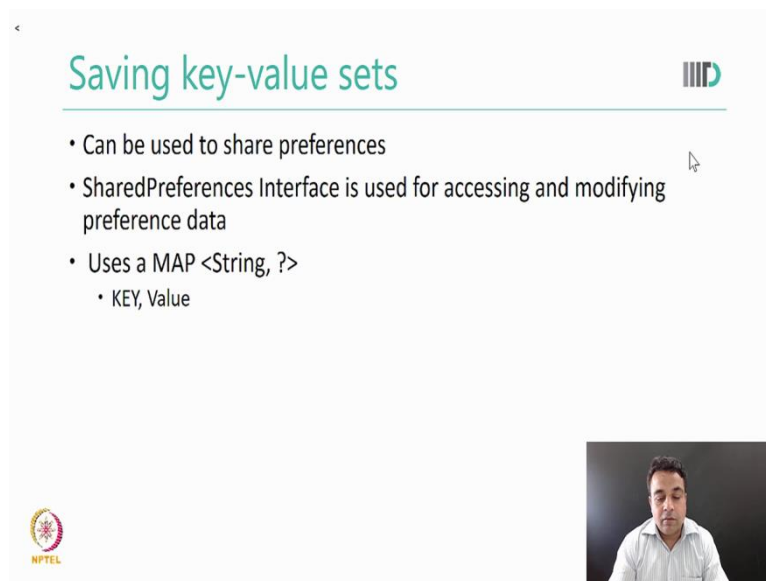
Data

- Apps need to save data
- Saving key-value pairs of simple data types in a shared preferences file
- Saving arbitrary files in Android's file system
- Using databases managed by SQLite

NPTEL

So as you see the motivation is very simple we need to save some data. There are 3 multiple ways 3 possible ways to do it, number one that you can save key value pairs of simple data types in a shared preference. Number two that you can save a file in the android file system and today we will learn about android file system in more detail. And number three you can use a very simple lite way database called SQLite. SQLite is based on SQL databases; however this is a version which is more suited for the mobile devices. So android provides us these three ways and today we are going to study them to see what is good in what scenario.

(Refer Slide Time: 2:03)



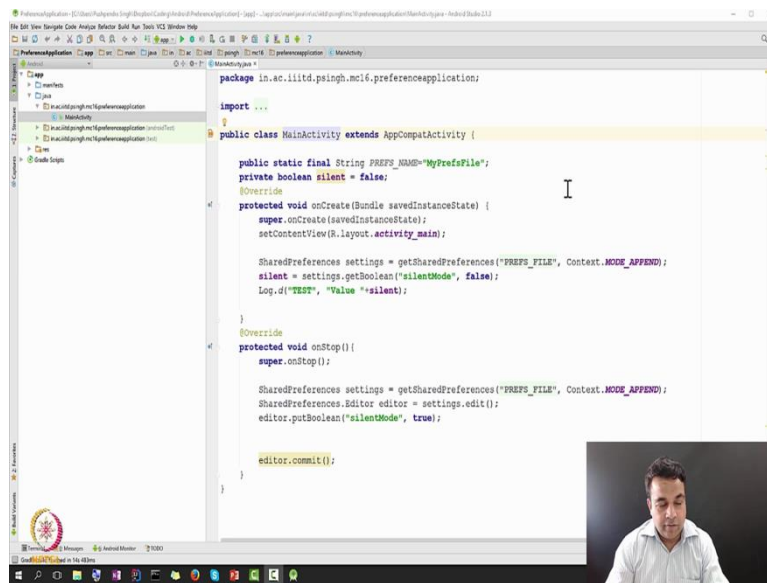
The slide is titled "Saving key-value sets" in a teal font. It features three bullet points: "Can be used to share preferences", "SharedPreferences Interface is used for accessing and modifying preference data", and "Uses a MAP <String, ?>" with a sub-bullet "KEY, Value". A mouse cursor points to the second bullet point. In the bottom right corner, there is a small video inset showing a man in a light blue shirt. The NPTEL logo is in the bottom left corner.

- Can be used to share preferences
- SharedPreferences Interface is used for accessing and modifying preference data
- Uses a MAP <String, ?>
 - KEY, Value

So let us start with the first one saving key value sets. This can be used to save this to save preferences there is a type here this is save preferences. The preferences that we save and later on which may also share are the preferences for example, suppose your application starts and you want to set a same volume level as the user selected last time or you want to start your application in the type of orientation that the user selected last time. So this kind of data is very useful for customizing your application and providing a personalized view of your application to the user. However, this data must be saved across different instantiation of your application. So this must be saved in such a way that it is not lost when your app is closed.

The android provides you a very simple way to save this data by preferences. It is nothing but a key value set, so internally it uses a MAP from java you know what a MAP is, in this case this MAP has a key of type String and then the value could be of any other type. And then using this MAP the values are stored and then using the key the values are retrieved. So this is the simplest possible way of storing some data in android. Let us quickly see an example on how it is done. So let me open an example that I have created. Let us first close our existing project, let us open another project.

(Refer Slide Time: 4:23)



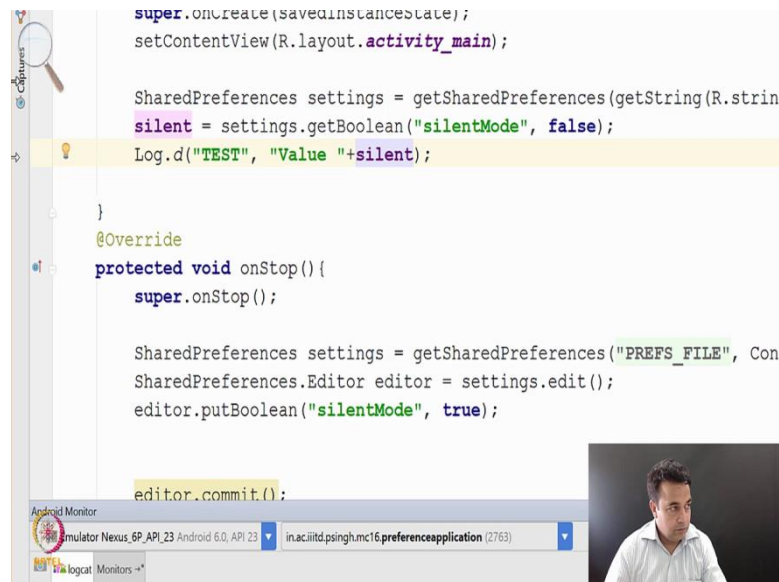
So this is a very simple application that I have developed only to show you the concept of shared preferences. This application has only one activity which I created using android studio. Now let us go through the code line by line, so the first line I declare are static final string which is giving nothing but just a name. So I call it PREFERENCES_NAME and I give it a name any name is fine. After that I declare a Boolean silent which I declare to be false. We will use this Boolean value to display our concept. What we want to show is that this value is preserved across different instantiation of the application. For us this value may represent that the user when interacting with your application puts the phone on the silent mode. Now let us see what we are creating in the onCreate folder.

So our onCreate is simple except that when we start our activity we read the shared preferences, so for that we use the function called getSharedPreferences, we give it a name and we give it a context as you can see here, so the name is the name given and then the context. And from that preferences file I do a get Boolean because I want the Boolean value and I set the silent value, so this will happen whenever my application runs. However for the very first time this needs also to be stored so we need to also add some data which gets run when your activity closes down.

So I have added some data in the onStop where I am using the same file and the same mode, but this time I am using an editor to edit. And using that editor I am using something called putBoolean() where I am setting the value of the silent mode Boolean value, so that my key is silent mode where I am setting a true to it while currently there is false. And hopefully once I have run my application once then it is said to be true. And then it will always return me the

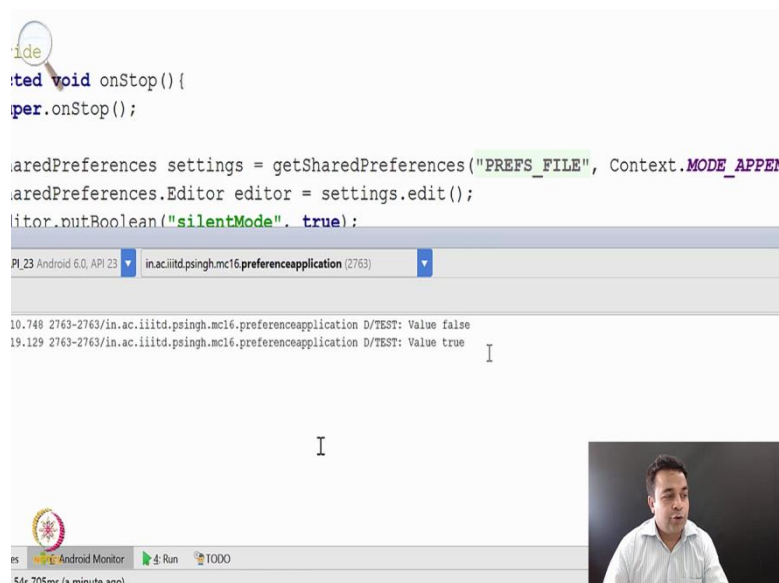
true and if there is some problem then it will return me the false. So let us run this program and see its execution and try to understand it. So I run the program we choose the same emulator that I choose always 6P.

(Refer Slide Time: 9:23)



As you see that let me just let me just increase its size let me just rotate it so that you can see it more easily. What we are looking for are these log messages, so our program is still starting let us take yes, so now our app is starting and let me set the filter as you will see that the initial value printed is false. This value has got printed because that initially it is false. Now let us close this application, let us make sure that the `onStop` is run and then we will restart the application and we will hope that it will read from the `SharedPreferences` file and this value will be different. So let us press the back button is set hopefully it trend on stop. Now I will go back to the same app and I will run it.

(Refer Slide Time: 10:16)



```

@Override
protected void onStop() {
    super.onStop();

    SharedPreferences settings = getSharedPreferences("PREFS_FILE", Context.MODE_PRIVATE);
    SharedPreferences.Editor editor = settings.edit();
    editor.putBoolean("silentMode", true);
}

```

Logcat output:

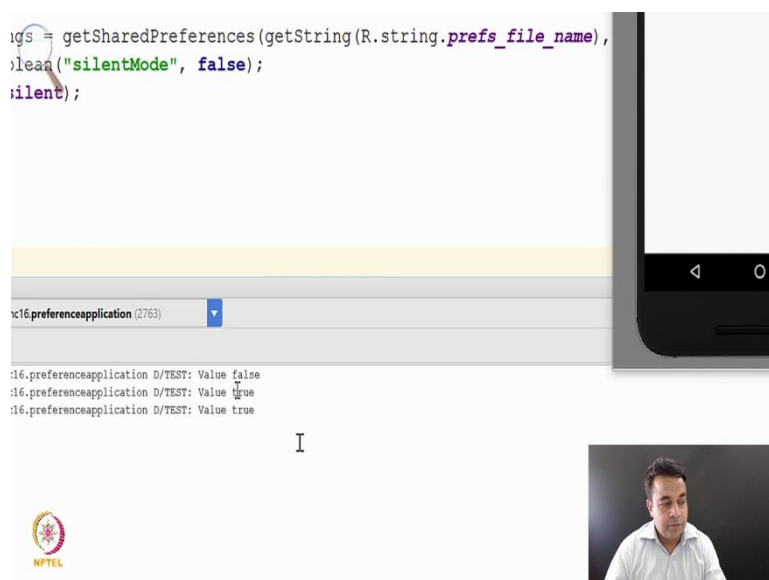
```

10.748 2763-2763/in.ac.iiitd.psingh.mc16.preferenceapplication D/TEST: Value false
19.129 2763-2763/in.ac.iiitd.psingh.mc16.preferenceapplication D/TEST: Value true

```

And now you can see that the value true is printed because last time when we exited the execution we shared it in the SharedPreferences our key was silent mode and our value was true and when we started it we retrieve the value of the silent mode and we printed it, so in the very first instance it was false but then it is true. Let us run it again and this time again we only have true.

(Refer Slide Time: 11:19)



```

SharedPreferences prefs = getSharedPreferences(getString(R.string.prefs_file_name),
    Context.MODE_PRIVATE);
prefs.putBoolean("silentMode", false);

```

Logcat output:

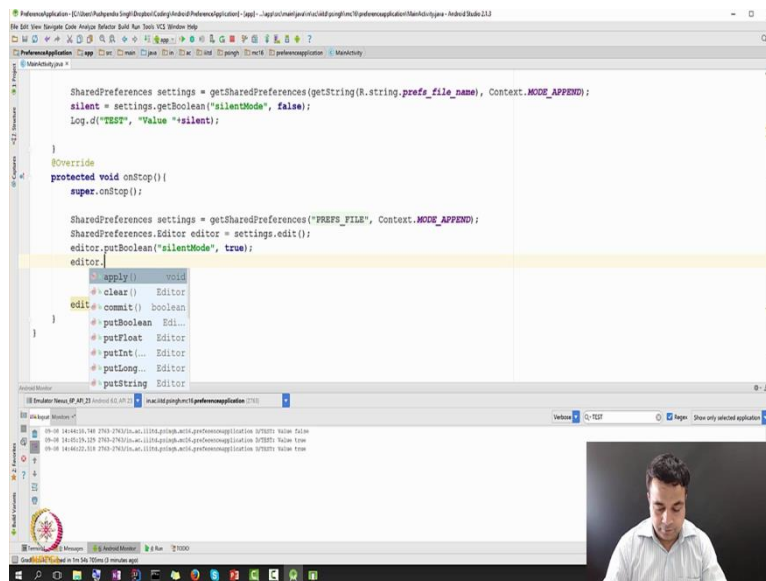
```

:16.preferenceapplication D/TEST: Value false
:16.preferenceapplication D/TEST: Value true
:16.preferenceapplication D/TEST: Value true

```

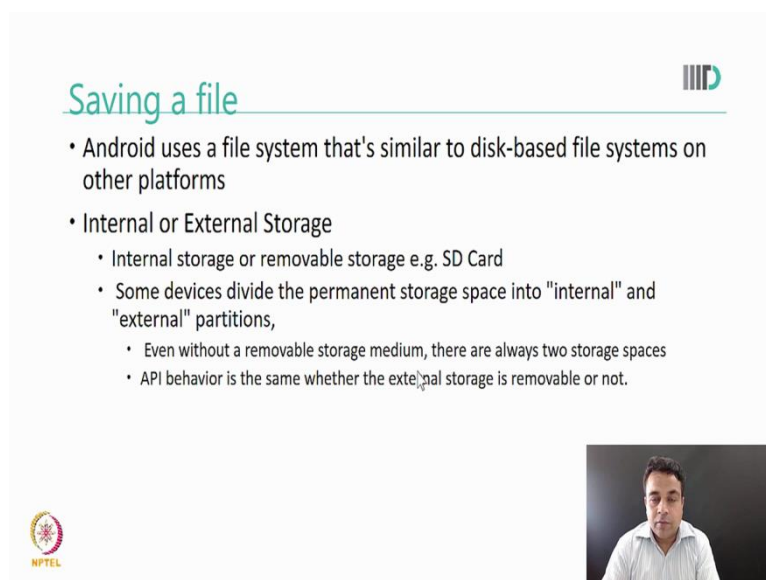
And now no matter how many times we stop and run it, it will be true because that is the value that we set (11:25). So this example, though very small source code, illustrates the concept of SharedPreferences very nicely. Let us go back to the code again, we do not do much we only use an editor and set the value by using the put method.

(Refer Slide Time: 11:49)



If you see we can have multiple put methods as you can see here we have putBoolean, putFloat, putInt, putLong, putString, etc, etc. Similarly we will have the get corresponding get methods for the same, so this is the easiest way in which you can store some data about your app persistently across different instantiation of your app. Now let us move to the second method which is more convenient when you want to store more data.

(Refer Slide Time: 12:31)



So saving a file, now many of you must be familiar and we discussed that in the earlier classes that android is primarily based on a Linux operating system. So internally android is nothing but a Linux operating system and that is why it uses a file system which is similar to the file system that you studied in your operating system course. However, there are some

differences because android once on a mobile device one thing which is very particular about mobile device is this concept of internal storage and external storage.

You may have seen that many mobile devices comes with this extra memory card slot, while the other mobile devices do not come, so this extra memory card slot is usually called an external storage and while the internal disk of the mobile is normally called an internal storage. However, to take advantage of this internal storage and external storage some devices even when they do not provide the extra memory cards slot divides their internal storage into two parts that is of internal storage and external storage. We will discuss that very shortly that why these devices do that and what is the advantage of it. However from the android side the API behavior remains same whether the external storage is a true external storage or it is just designated by the device. Now let us see some advantages or disadvantages or let us say properties of internal storage and external storage.

(Refer Slide Time: 14:28)

Storage

- It's always available.
- Files saved here are accessible by only your app.
- When the user uninstalls your app, the system removes all your app's files from internal storage.

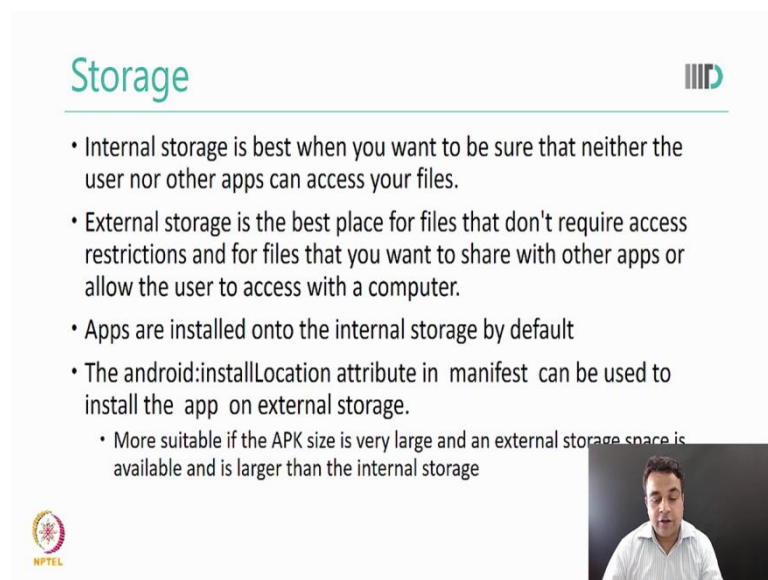
- The user can mount the external storage as USB storage and in some cases remove it from the device.
- It's world-readable, so files saved here may be read outside of your control.
- When the user uninstalls your app, the system removes your app's files from here only if you save them in the `getExternalFilesDir`

So, on the left side it is the internal and on the right side it is the external. So the internal storage as you know is always available, that is you are always sure that there is some space available. Even the basic android devices have some storage available for you. On the other hand, the same assumption cannot be made about the external storage. Number one your phone may not have an external storage at all that is it may not have a slot for the memory card or even if it has when you connect your phone to your PC for example, and you mount the external storage as USB you cannot use it. So the assumption that it is always available is not true for external storage. Please keep that in mind when you write the code. Number two,

for internal storage whatever you save in the internal storage of an app is accessible only to the app. On the contrary, whatever you store in the external storage can be read by anywhere.

So be aware of this when you are developing your application. The third major difference is that when a user uninstalls an app then the system removes all your app files from the internal storage. However, when the user uninstalls your app and your app has stored something on the external storage only those files will be removed which have been created with using a specific API rest of the files will be left on the external storage. Now we will come and discuss it in more detail later on but take a pause and think about the benefits as well as the disadvantages of this approach that is some of the files will be removed while the others will not be removed, ok.

(Refer Slide Time: 17:01)



The slide is titled "Storage" in a teal font. It features a list of five bullet points. The first four points are standard, while the fifth point has a sub-bullet. In the bottom right corner, there is a small video inset showing a man speaking. The NPTEL logo is in the bottom left corner.

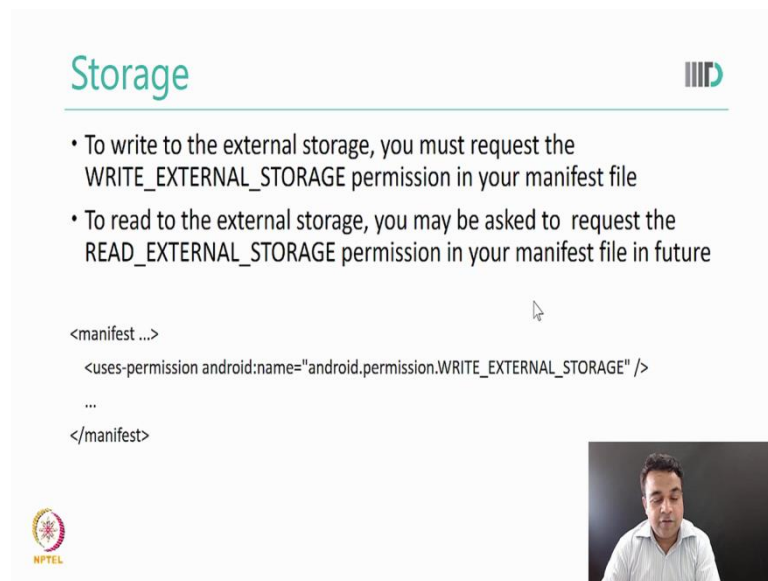
- Internal storage is best when you want to be sure that neither the user nor other apps can access your files.
- External storage is the best place for files that don't require access restrictions and for files that you want to share with other apps or allow the user to access with a computer.
- Apps are installed onto the internal storage by default
- The android:installLocation attribute in manifest can be used to install the app on external storage.
 - More suitable if the APK size is very large and an external storage space is available and is larger than the internal storage

So depending on these major characteristics or properties we can say that internal storage is best when you want to be sure that neither the user nor other apps can access your files, so only your app can access the files. At the same time, external storage is the best place for files which do not require access restrictions for example, if you have developed an app which takes photos using the camera. Now you may want your user to see these pictures and may be also to share them with other users. So for this external storage is the best, by default your app are installed on the internal storage only.

But you can use install Location attribute in the manifest to make sure your app is installed on the external storage. So particularly if your APK is very large you may want to make the switch because if you consume on a large part of internal storage of a user they are not going

to use it. Now considering these advantages can you make a guess that why some devices designate their internal storage as internal and external, what do you think could be the advantage of it? So as you may guess one advantage is that now the app can decide which files are used even after the app is uninstalled and which files to be removed when the app is uninstalled. For example, if your app is about taking pictures you may not want to remove the photos that have already been taken and the easiest way to do it is to store them in the external storage.

(Refer Slide Time: 19:24)



The slide is titled "Storage" in a teal font. It contains two bullet points: "To write to the external storage, you must request the WRITE_EXTERNAL_STORAGE permission in your manifest file" and "To read to the external storage, you may be asked to request the READ_EXTERNAL_STORAGE permission in your manifest file in future". Below the text is a code snippet for an Android manifest file. In the bottom right corner, there is a small video feed of a man speaking. The NPTEL logo is in the bottom left corner.

- To write to the external storage, you must request the WRITE_EXTERNAL_STORAGE permission in your manifest file
- To read to the external storage, you may be asked to request the READ_EXTERNAL_STORAGE permission in your manifest file in future

```
<manifest ...>  
  <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />  
  ...  
</manifest>
```



Now if you want to write to an external storage you must request a specific permission in your manifest file which is of type WRITE_EXTERNAL_STORAGE. Similarly for reading even though it is not currently necessary but you may want need permission in future for doing even the reading of the external storage? This permission is READ_EXTERNAL_STORAGE. My advice to you is that no matter what app you may develop please ask for these permissions, so that your application remains future proof.

This can be set very easily in your manifest file here I am giving you a very simple example user permission android name android.permission.WRITE_EXTERNAL_STORAGE. If you do that when user installs your application if it is on an older device holding android 5.0 or older version then user will be ask for the permission at the install time, if it is a new device then the user will be asked for the permission in the run time.

(Refer Slide Time: 20:31)

Saving a File on Internal Storage

- `getFilesDir()`
 - Returns a File representing an internal directory for your app.
- `getCacheDir()`
 - Returns a File representing an internal directory for your app's temporary cache files.
 - Be sure to delete each file once it is no longer needed and implement a reasonable size limit for the amount of memory you use at any given time, such as 1MB.
 - If the system begins running low on storage, it may delete your cache files without warning.





Now let us see that how do we save a file on the internal storage. The first step that we need to do is to find out the directory that corresponds to our application. We can get this directory by making a method call name as `getFilesDirectory()`. Similarly, there is another method call available for us which says that `getCacheDirectory()` and returns a file representing an internal directory for our apps temporary cache files. Please do not confuse this cache with the operating system cache, these two are different. The cache of an application is the temporary cache file consists of files which the app uses. Now be sure to delete each file once it is no longer needed and implement a reasonable size limit for the amount of memory you use at any given time because if the system begins running low on storage, it may delete your cache files without warning, do not rely on the cache files.

(Refer Slide Time: 21:49)

Saving a File on Internal Storage

```
File file = new File(context.getFilesDir(), filename);  
  
createTempFile()
```





Here is a very simple example, good part of android programming is that it is very similar to java programming as we have already seen. So just like in java once you get a file object you can write to the file in multiple ways. Similarly in android once you get a file object you can write the file in multiple ways. For example, here I am getting a file object, once I get the file object from the required directory and by giving filename I can write to the file in multiple ways.

(Refer Slide Time: 22:23)

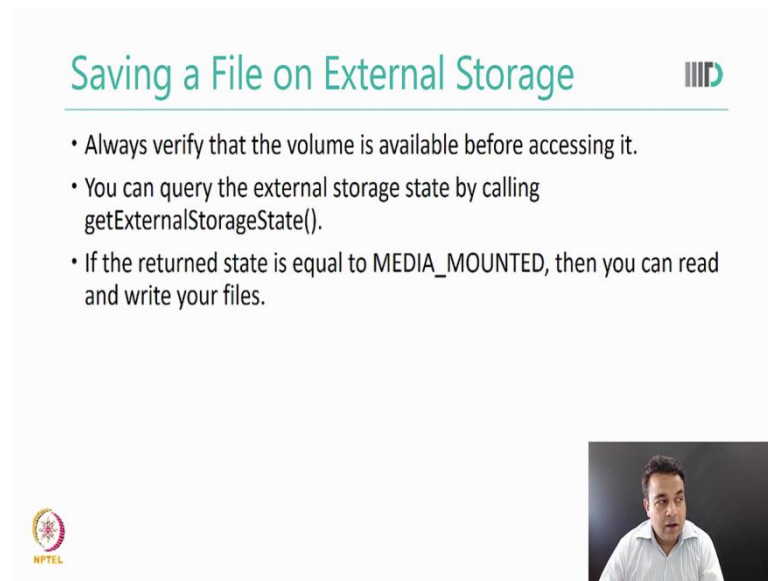
Saving a File on Internal Storage

```
String filename = "myfile";  
String string = "Hello world!";  
FileOutputStream outputStream;  
  
try {  
    outputStream = openFileOutput(filename, Context.MODE_PRIVATE);  
    outputStream.write(string.getBytes());  
    outputStream.close();  
} catch (Exception e) {  
    e.printStackTrace();  
}
```



So here is a very simple example where I am using output stream to write to my file. Now that was all about internal storage now let us discuss the external storage.

(Refer Slide Time: 22:39)



The slide is titled "Saving a File on External Storage" in a teal font. It features a list of three bullet points: "Always verify that the volume is available before accessing it.", "You can query the external storage state by calling `getExternalStorageState()`.", and "If the returned state is equal to `MEDIA_MOUNTED`, then you can read and write your files." In the bottom right corner, there is a small video inset showing a man speaking. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it.





- Always verify that the volume is available before accessing it.
- You can query the external storage state by calling `getExternalStorageState()`.
- If the returned state is equal to `MEDIA_MOUNTED`, then you can read and write your files.

So as told earlier that there is no guarantee that the external storage will be available. And because there is no guarantee that the external storage will be available, you must always verify that volume is available before. Now you can query the external storage by calling a simple method called `getExternalStorageState()` and if the returned state is equal to `MEDIA_MOUNTED`, then you can read and write your files. But this verification step is very important because you can never be sure that your external storage available specially when your app is going to be run on different types of devices, some of them may have external storage some of them may not, some of them which may have an external storage may be connected to the PC with USB mounted as a scenario so at that time it will not be available, so it is always just go to verify.

(Refer Slide Time: 23:49)

Saving a File on External Storage

```
/* Checks if external storage is available for read and write */
public boolean isExternalStorageWritable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state)) {
        return true;
    }
    return false;
}
```









Here is a very simple code which will do it for you. So is ExternalStorageWritable() state getExternalStorageState() if it is a MEDIA_MOUNTED then return true otherwise return false. Use this method as explained earlier this will be the state. Here we are checking for the same and then you can write on your external storage.

(Refer Slide Time: 24:20)

Saving on External Storage

- Public files
 - Files that should be freely available to other apps and to the user. When the user uninstalls your app, these files should remain available to the user.
 - For example, photos captured by your app or other downloaded files.
- Private files
 - Files that rightfully belong to your app and should be deleted when the user uninstalls your app.
 - Although these files are technically accessible by the user and other apps because they are on the external storage, they are files that realistically don't provide value to the user outside your app.
 - When the user uninstalls your app, the system deletes all files in your private directory.
 - For example, additional resources downloaded by your app or temporary files.



Now more details about the external storage, external storage has two types of files. One is public files and the other is private files. So the public files as shown by the name are the files that should be freely available to other apps and to the user. So when the user uninstalls your app these files should remain available to the user. For example, photos captured by your app or other downloaded files.

Then you may also want to use the external storage to store some private files. And these files should belong to your app and should be deleted when the user uninstalls your app. Now please note a crucial difference, when the files are on the internal storage no other user can access it, but when they are on the external storage they can be accessed. So even though we are saving private files they are not really private, they are only private in the sense that they are not useful beyond your application.

So these files are technically accessible but ideally they should not be the type of files that are of (()) (25:34). And when the user uninstalls your app the system should delete all these all such files. So an example of such files is additional resources downloaded by your app or temporary media files. For example, when you are making math quiz we had these small icons now you may be you may want to have 10-20 different type of icons for your app, these icons have no meaning outside your app. So you may want to use the external storage to save this but then the all of them should be deleted if the user deletes your app.

(Refer Slide Time: 26:17)



Saving a Public File

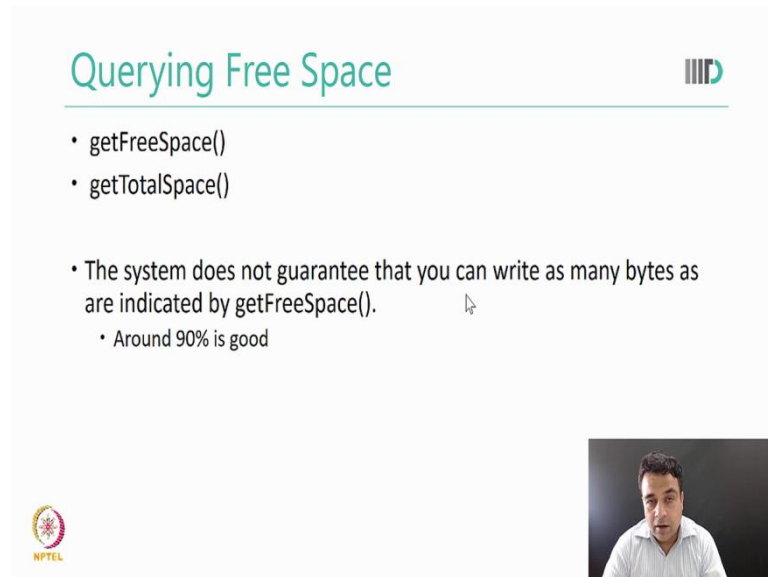
```
public File getAlbumStorageDir(String albumName) {  
    // Get the directory for the user's public pictures directory.  
    File file = new File(Environment.getExternalStoragePublicDirectory(  
        Environment.DIRECTORY_PICTURES), albumName);  
    if (!file.mkdirs()) {  
        Log.e(LOG_TAG, "Directory not created");  
    }  
    return file;  
}
```

The slide includes a small video inset of a man in a white shirt speaking, and a bottom bar with various icons including NPTEL.

Now let us look at a small code to see how to save a public file. Here I am trying to do that I try to get a file object new File. Please look at this variable very carefully; I am saying DIRECTORY_PICTURES so I am trying to get the directory for the user public pictures directory. There are different types defined in android and what they help is in finding what kind of files they are so that the android can use certain apps to display it. Suppose your app uses a code something like this, then android can use its own knowledge to find out that ok these files looks like picture files and therefore may be they can be displayed in the (())

(27:10). Similar things will be available for example for a dial tone or for music files, etc, so try to use these standards when you are developing your code.

(Refer Slide Time: 27:32)



The slide is titled "Querying Free Space" in a teal font. It features a list of methods and a warning. The methods listed are `getFreeSpace()` and `getTotalSpace()`. A warning states that the system does not guarantee that you can write as many bytes as are indicated by `getFreeSpace()`, and suggests that around 90% is good. The slide also includes the NPTEL logo in the bottom left and a small video feed of a presenter in the bottom right.

Querying Free Space

- `getFreeSpace()`
- `getTotalSpace()`
- The system does not guarantee that you can write as many bytes as are indicated by `getFreeSpace()`.
 - Around 90% is good



Querying for free space again besides knowing that your external storage available you must have also query, whether it has the space or not. You can use either of the two methods, first is a `getFreeSpace()` and other is a `getTotalSpace()`. As their name indicate first will see the free space available and another we will see the total space available, often they are used together. But be aware that the free space if it returns the size which you just exactly need for your file it may not actually be sufficient. So if you want to store something of 10 MB and the `getFreeSpace()` becomes 10 MB your android system still refuse it. So a rule of the thumb is that around 90 percent of whatever is return can be used by it, so if it returns 10 MB around 9 MB you can use.

(Refer Slide Time: 28:27)

Deleting a File

- `myFile.delete();`
- `myContext.deleteFile(fileName);`

- When the user uninstalls your app, the Android system deletes the following:
 - All files you saved on internal storage
 - All files you saved on external storage using `getExternalFilesDir()`.



Then the next is deleting a file again it is very easy you just call a method call `delete` or `deleteFile` and pass the file name then it deletes the file. Similarly as told earlier, if the user uninstalls your app then android system will delete. Number one all files you saved on internal storage and all files you saved on external storage using `ExternalFilesDirectory()`. This was all about files which are one of the major ways of creating and storing data from your application, next we will learn about the database called SQL.