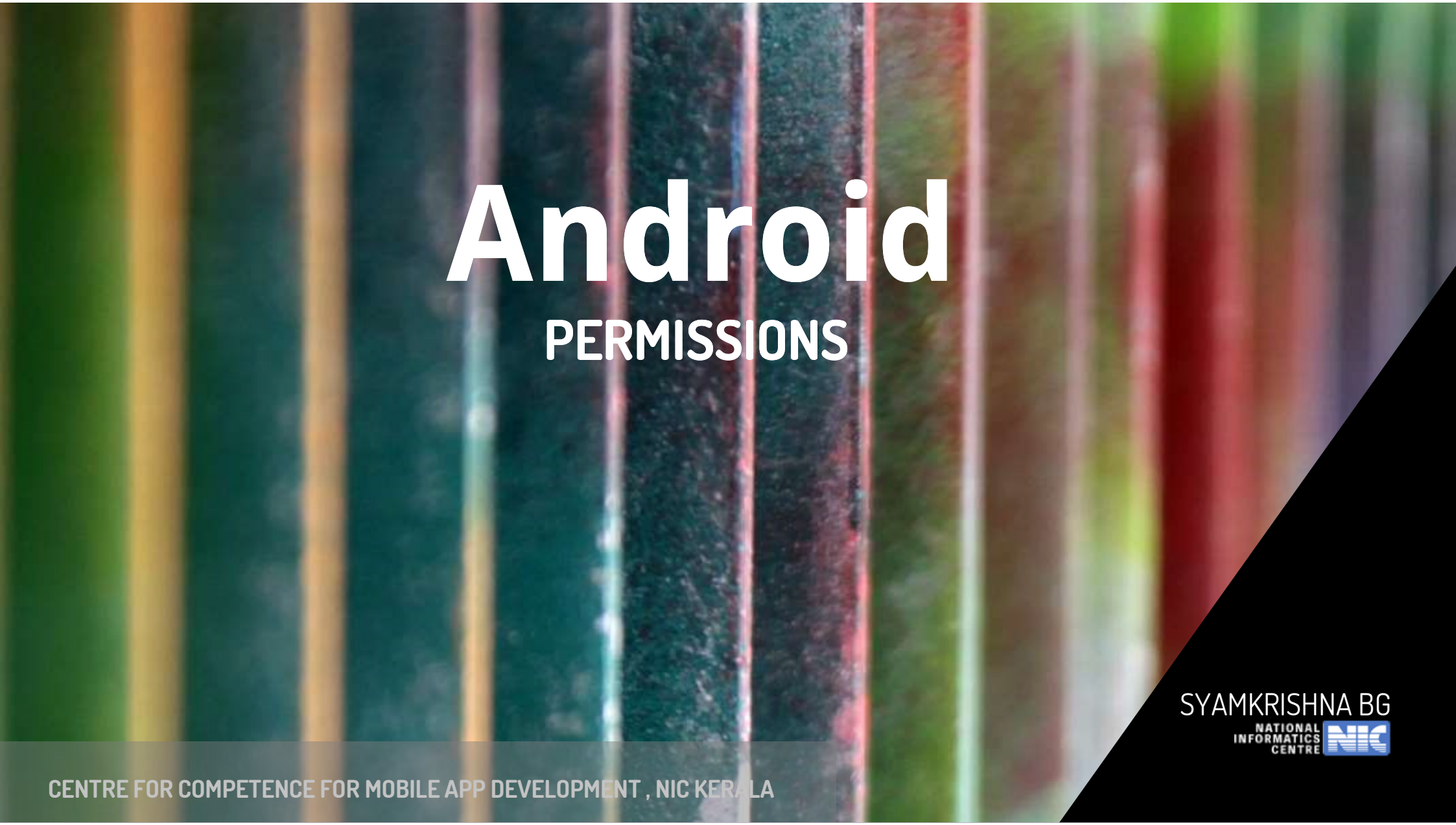


SYAMKRISHNA BG
NATIONAL INFORMATICS CENTRE
CENTRE FOR COMPETENCE FOR MOBILE APP DEVELOPMENT , NIC KERALA

Introduction

The purpose of a permission is to protect the privacy of an Android user. Android apps must request permission to access sensitive user data (such as contacts and SMS), as well as certain system features (such as camera and internet). Depending on the feature, the system might grant the permission automatically or might prompt the user to approve the request.

An app must publicize the permissions it requires by including <uses-permission> tags in the app manifest

Example

For example, an app that needs to send SMS messages would have this line in the manifest:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
        package="in.nic.kerala.training">

                <uses-permission android:name="android.permission.SEND_SMS"/>

    <application ...>
        ...
    </application>

</manifest>
```

Example

With respect to web service calls

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
        package="in.nic.kerala.training">

    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
    <uses-permission android:name="android.permission.INTERNET" />

    <application ...>
        ...
    </application>

</manifest>
```

What if the requested hardware component is not available?

<**uses-feature** android:name="android.hardware.camera" android:required="false" />

If you declare android:required="false" for the feature, then Google Play allows your app to be installed on devices that don't have the feature.

If you don't provide the **<uses-feature> tag**, then when Google Play sees that your app requests the corresponding permission, it assumes your app requires this feature. So it filters your app from devices without the feature, as if you declared android:required="true" in the <uses-feature> tag.

Protection levels

Permissions are divided into several protection levels. The protection level affects whether runtime permission requests are required.
There are three protection levels that affect third-party apps:

normal,
signature,
dangerous

Protection levels NORMAL

As of Android 8.1 (API level 27), the following permissions are classified as PROTECTION_NORMAL:

ACCESS_LOCATION_EXTRA_COMMANDS
ACCESS_NETWORK_STATE
ACCESS_NOTIFICATION_POLICY
ACCESS_WIFI_STATE
BLUETOOTH
BLUETOOTH_ADMIN
BROADCAST_STICKY
CHANGE_NETWORK_STATE
CHANGE_WIFI_MULTICAST_STATE
CHANGE_WIFI_STATE
DISABLE_KEYGUARD
EXPAND_STATUS_BAR
GET_PACKAGE_SIZE
INSTALL_SHORTCUT
INTERNET
KILL_BACKGROUND_PROCESSES
MANAGE_OWN_CALLS
MODIFY_AUDIO_SETTINGS
NFC
READ_SYNC_SETTINGS
READ_SYNC_STATS
RECEIVE_BOOT_COMPLETED
REORDER_TASKS
REQUEST_COMPANION_RUN_IN_BACKGROUND
REQUEST_COMPANION_USE_DATA_IN_BACKGROUND
REQUEST_DELETE_PACKAGES
REQUEST_IGNORE_BATTERY_OPTIMIZATIONS
REQUEST_INSTALL_PACKAGES
SET_ALARM
SET_WALLPAPER
SET_WALLPAPER_HINTS
TRANSMIT_IR
USE_FINGERPRINT
VIBRATE
WAKE_LOCK
WRITE_SYNC_SETTINGS

Protection levels DANGEROUS

Permission Group	Permissions
CALENDAR	• READ_CALENDAR • WRITE_CALENDAR
CAMERA	• CAMERA
CONTACTS	• READ_CONTACTS • WRITE_CONTACTS • GET_ACCOUNTS
LOCATION	• ACCESS_FINE_LOCATION • ACCESS_COARSE_LOCATION
MICROPHONE	• RECORD_AUDIO
PHONE	• READ_PHONE_STATE • READ_PHONE_NUMBERS • CALL_PHONE • ANSWER_PHONE_CALLS • READ_CALL_LOG • WRITE_CALL_LOG • ADD_VOICEMAIL • USE_SIP • PROCESS_OUTGOING_CALLS
SENSORS	• BODY_SENSORS
SMS	• SEND_SMS • RECEIVE_SMS • READ_SMS • RECEIVE_WAP_PUSH • RECEIVE_MMS
STORAGE	• READ_EXTERNAL_STORAGE • WRITE_EXTERNAL_STORAGE

From Android 6.0 only dangerous permissions are checked at runtime, normal permissions are not.

Protection levels SIGNATURE

As of Android 8.1 (API level 27), the following permissions that third-party apps can use are classified asPROTECTION_SIGNATURE:

BIND_ACCESSIBILITY_SERVICE
BIND_AUTOFILL_SERVICE
BIND_CARRIER_SERVICES
BIND_CHOOSER_TARGET_SERVICE
BIND_CONDITION_PROVIDER_SERVICE
BIND_DEVICE_ADMIN
BIND_DREAM_SERVICE
BIND_INCALL_SERVICE
BIND_INPUT_METHOD
BIND_MIDI_DEVICE_SERVICE
BIND_NFC_SERVICE
BIND_NOTIFICATION_LISTENER_SERVICE
BIND_PRINT_SERVICE
BIND_SCREENING_SERVICE
BIND_TELECOM_CONNECTION_SERVICE
BIND_TEXT_SERVICE
BIND_TV_INPUT
BIND_VISUAL_VOICEMAIL_SERVICE
BIND_VOICE_INTERACTION
BIND_VPN_SERVICE
BIND_VR_LISTENER_SERVICE
BIND_WALLPAPER
CLEAR_APP_CACHE
MANAGE_DOCUMENTS
READ_VOICEMAIL
REQUEST_INSTALL_PACKAGES
SYSTEM_ALERT_WINDOW
WRITE_SETTINGS
WRITE_VOICEMAIL

Run Time Permission Sample Code

```
if (!checkPermission()) {

    requestPermission();

} else {

    Snackbar.make(view, "Permission already granted.", Snackbar.LENGTH_LONG).show();

}
```

```java
private boolean checkPermission() {
    int result = ContextCompat.checkSelfPermission(getApplicationContext(), ACCESS_FINE_LOCATION);
    int result1 = ContextCompat.checkSelfPermission(getApplicationContext(), CAMERA);
    return result == PackageManager.PERMISSION_GRANTED && result1 == PackageManager.PERMISSION_GRANTED;
}

private void requestPermission() {

    ActivityCompat.requestPermissions(this, new String[]{ACCESS_FINE_LOCATION, CAMERA}, PERMISSION_REQUEST_CODE);

}
```

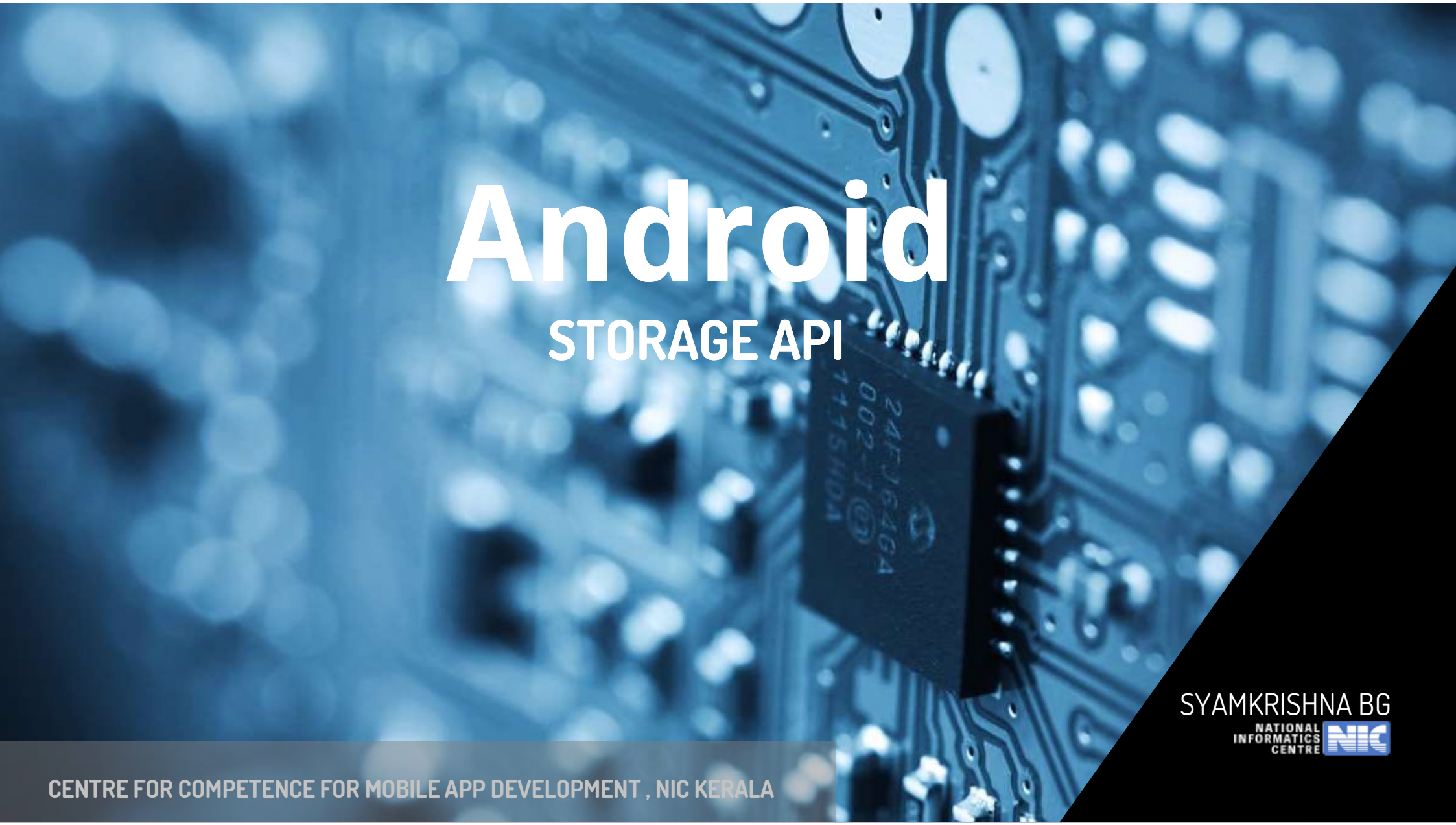
```java
@Override
public void onRequestPermissionsResult(int requestCode, String permissions[], int[] grantResults) {
    switch (requestCode) {
        case PERMISSION_REQUEST_CODE:
            if (grantResults.length > 0) {

                boolean locationAccepted = grantResults[0] == PackageManager.PERMISSION_GRANTED;
                boolean cameraAccepted = grantResults[1] == PackageManager.PERMISSION_GRANTED;

                if (locationAccepted && cameraAccepted)
                    Snackbar.make(view, "Permission Granted, Now you can access location data and camera.", Snackbar.LENGTH_LONG).show();
                else {
                    Snackbar.make(view, "Permission Denied, You cannot access location data and camera.", Snackbar.LENGTH_LONG).show();

                    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
                        if (shouldShowRequestPermissionRationale(ACCESS_FINE_LOCATION)) {
                            showMessageOKCancel("You need to allow access to both the permissions",
                                    new DialogInterface.OnClickListener() {
                                        @Override
                                        public void onClick(DialogInterface dialog, int which) {
                                            if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
                                                requestPermissions(new String[]{ACCESS_FINE_LOCATION, CAMERA},
                                                        PERMISSION_REQUEST_CODE);
                                            }
                                        }
                                    });
                            return;
                        }
                    }

                }
            }
            break;
    }
}
```



Android

STORAGE API

SYAMKRISHNA BG
NATIONAL INFORMATICS CENTRE

CENTRE FOR COMPETENCE FOR MOBILE APP DEVELOPMENT , NIC KERALA

Introduction

Android provides several options to presist data privately or publicly
Selection of storage option depends on Data Type, Size, LifeSpan & Scope

Data Type :

- Primitive / Non Primitive
- Media Files
- Relational Data

Lifespan :

- App Execution Time
- App Lifetime
- After App Removal (uninstall)

Scope :

- To Specific App Component
- Among App Components
- To Specific Apps
- To all

Android Local Storage

- Internal file storage
- External file storage
- Shared Preference
- Database

Internal File Storage

•Files saved to the internal storage are private to your app

•The system provides a private directory on the file system for each app where you can organize any files your app needs.

•On Android 6.0 (API level 23) and lower, other apps can read your internal files if you set the file mode to be world readable. However, the other app must know your app package name and file names.

•your app does not require any system permissions to read and write to the internal directories

Internal Storage Options Provided

- **App Specific Internal Storage**
- **Common Cache Storage**

Example Storage-App Specific Internal File Storage

```
String filename = "myfile";
String fileContents = "Hello world!";
FileOutputStream   outputStream;

try
{
   outputStream = openFileOutput(filename, Context.MODE_PRIVATE );
   outputStream.write(fileContents.getBytes());
   outputStream.close();
}
catch (Exception e)
{
   e.printStackTrace();
}
```

Know
Context.MODE_PRIVATE

Example Storage-App Specific Cache Storage

```
private File getTempFile(Context context, String url) {
    File file;
    try {
        String fileName = Uri.parse(url).getLastPathSegment();
        file = File.createTempFile (fileName, null, context.getCacheDir());
    } catch (IOException e) {
        // Error while creating file
    }
    return file;
}
```

If the system runs low on storage, it may delete your cache files without warning, so make sure you check for the existence of your cache files before reading them.

Files created with createTempFile() are placed in a cache directory that's private to your app

External Storage

•it is a storage space that users can mount to a computer as an external storage device, and it might even be physically removable (such as an SD card).
•Files saved to the external storage are world-readable and can be modified by the user
•To write to the public external storage, you must request the WRITE_EXTERNAL_STORAGE permission in your manifest file

```
<manifest ...>

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

```
<manifest ...>
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"
                     android:maxSdkVersion="18" />
    ...
</manifest>
```

External Storage- Save to a public directory

```
public File getPublicAlbumStorageDir(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;
}
```

Environment.**DIRECTORY_PICTURES**
Standard directory in which to place pictures that are available to the user. Note that this is primarily a convention for the top-level public directory, as the media scanner will find and collect pictures in any directory.

DIRECTORY_MUSIC,DIRECTORY_RINGTONES etc.

External Storage- Save to a private directory

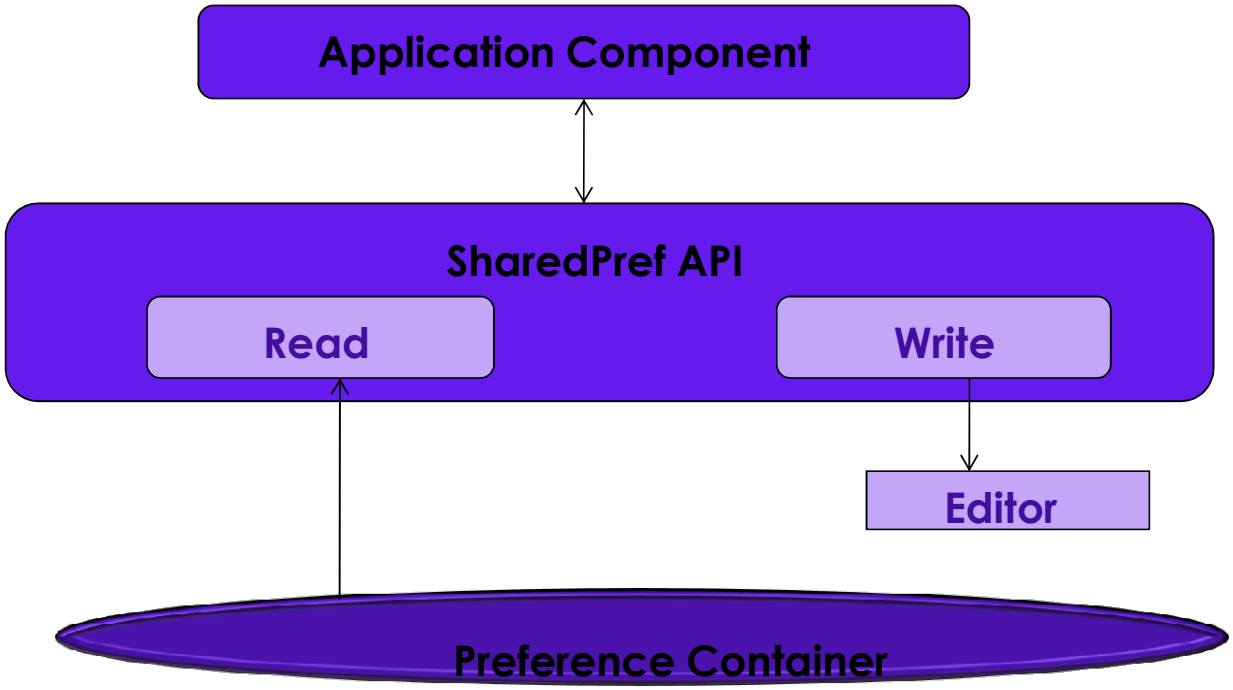
```
public File getPrivateAlbumStorageDir(Context context, String albumName) {
    // Get the directory for the app's private pictures directory.
    File file = new File(context.getExternalFilesDir(Environment.DIRECTORY_PICTURES), albumName);
    if (!file.mkdirs()) {
        Log.e(LOG_TAG, "Directory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

Environment.**DIRECTORY_PICTURES**
Standard directory in which to place pictures that are available to the user. Note that this is primarily a convention for the top-level public directory, as the media scanner will find and collect pictures in any directory.

DIRECTORY_MUSIC,DIRECTORY_RINGTONES etc.

Shared preferences

If you don't need to store a lot of data and it doesn't require structure, you should use SharedPreferences. The SharedPreferences APIs allow you to read and write persistent key-value pairs
Supports primitive data types: **booleans, floats, ints, longs, and strings.**



SharedPreferences Storage

Create once

```
Context context = getActivity();
SharedPreferences sharedPref = context.getSharedPreferences
(
"in.nic.kerala.training.PREFERENCE_FILE_KEY", Context.MODE_PRIVATE
);
```

Write/Update many times using the KEY

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);
SharedPreferences.Editor editor = sharedPref.edit();
editor.putInt("KEY",VALUE);
editor.commit();
```

Read many times using the KEY

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);
int defaultValue = getResources().getInteger(R.integer.saved_high_score_default_key);
int highScore = sharedPref.getInt(getString(R.string.saved_high_score_key), defaultValue);
```

Note:
Context.MODE_PRIVATE

Features

Store data in the form of key-value pairs
Lightweight

Size : Small data e.g. app settings & flags

Lifespan : Till un installation / clear application data through settings

Methods

getSharedPreferences(name,MODE)

- used from within your Activity (or other application Context), to access application-level preferences

getSharedPreferences (String PREFS_NAME, int mode)

- **PREFS_NAME** is the name of the file.
- **mode** is the operating mode
  eg: MODE_PRIVATE,
  MODE_WORLD_WRITEABLE
  MODE_WORLD_READABLE

Write Operation :

```
Sharedpreferences sp = getSharedPreferences(MyPREFERENCES,
                        Context.MODE_PRIVATE);

SharedPreferences.Editor editor = sp.edit();

editor.putInt("KEY_INT",10);

editor.putString( "KEY_STR" , "Hello World !!" );

editor.commit(); // save changes

editor.remove("KEY_INT");          // remove specific key-value
editor.clear();                    // to flush shared pref completely
```

Read Operation :

```
SharedPreferences sp = getSharedPreferences( “my_prefs” , Context.MODE_PRIVATE );
Int value_int = sp.getInt(“KEY_INT” , 0);
String value_str = sp.getString(“KEY_STR” , null);
// second param = default value
```

SharedPreferences Commonly Used API Methods

putBoolean(String key, boolean value)
Set a boolean value in the preferences editor.

putFloat(String key, float value)
Set a float value in the preferences editor.

putInt(String key, int value)
Set a integer value in the preferences editor.

putLong(String key, long value)
Set a long value in the preferences editor.

putString(String key, String value)
Set a String value in the preferences editor.

remove(String key)
Mark in the editor that a preference value should be removed.

SharedPreferences Commonly Used API Methods

apply()
Commit your preferences changes back from this Editor to the SharedPreferences object.
clear()
Mark in the editor to remove all values from the preferences.
commit()
Commit your preferences changes back from this Editor to the SharedPreferences object.
getBoolean(String key, boolean value)
Set a boolean value in the preferences editor.
getFloat(String key, float value)
Set a float value in the preferences editor.
getInt(String key, int value)
Set a integer value in the preferences editor.
getLong(String key, long value)
Set a long value in the preferences editor.
getString(String key, String value)
Set a String value in the preferences editor.

Local Database Storage

SQL Lite

Android KEYSTORE

The Android Keystore system lets you store cryptographic keys in a container to make it more difficult to extract from the device. Once keys are in the keystore, they can be used for cryptographic operations with the key material remaining non-exportable. Moreover, it offers facilities to restrict when and how keys can be used, such as requiring user authentication for key use or restricting keys to be used only in certain cryptographic modes

https://developer.android.com/training/articles/keystore

NATIONAL INFORMATICS CENTRE

Thank you