# SNS COLLEGE OF TECHNOLOGY

**Coimbatore-35**
**An Autonomous Institution**

Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A++' Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

## DEPARTMENT OF INFORMATION TECHNOLOGY

## 19ITE310 - MOBILE APPLICATION DEVELOPMENT

III YEAR - VI SEM

UNIT 3 – **BUILDING BLOCKS OF MOBILE APPS – II**

TOPIC 1 – Broadcast receivers - Telephony and SMS APIs

# UNIT – 3
# BUILDING BLOCKS OF MOBILE APPS – I

Broadcast receivers - Telephony and SMS APIs - Native data handling – on-device file I/O – shared preferences - mobile databases such as SQLite, and enterprise data access (via Internet/Intranet)

## Lab Experiments:

1. Create a user registration application that stores the user details in a database table
2. Create an app for hospital management system for storing and retrieving patient records

# Android - Broadcast Receivers

- **Broadcast Receivers** simply respond to broadcast messages from other applications or from the system itself.
- These messages are sometime called events or intents.
- For example, applications can also initiate broadcasts to let other applications know that some data has been downloaded to the device and is available for them to use, so this is broadcast receiver who will intercept this communication and will initiate appropriate action.
- There are following two important steps to make BroadcastReceiver works for the system broadcasted intents −

. Creating the Broadcast Receiver.

. Registering Broadcast Receiver

- There is one additional step in case you are going to implement your custom intents then you will have to create and broadcast those intents.

# Android - Broadcast Receivers

- Creating the Broadcast Receiver
- A broadcast receiver is implemented as a subclass of BroadcastReceiver class and overriding the onReceive() method where each message is received as a Intent object parameter.

```
public class MyReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        Toast.makeText(context, "Intent Detected.", Toast.LENGTH_LONG).show();
    }
}
```

# Android - Broadcast Receivers

- Registering Broadcast Receiver

- An application listens for specific broadcast intents by registering a broadcast receiver in AndroidManifest.xml file.
- Consider we are going to register MyReceiver for system generated event ACTION_BOOT_COMPLETED which is fired by the system once the Android system has completed the boot process.

```
<application
  android:icon="@drawable/ic_launcher"
  android:label="@string/app_name"
  android:theme="@style/AppTheme" >
  <receiver android:name="MyReceiver">
       <intent-filter>
       <action android:name="android.intent.action.BOOT_COMPLETED">
       </action>
    </intent-filter>
    </receiver>
</application>
```

# Android - Broadcast Receivers

- Now whenever your Android device gets booted, it will be intercepted by BroadcastReceiver MyReceiver and implemented logic inside onReceive() will be executed.
- There are several system generated events defined as final static fields in the Intent class.
- The following table lists a few important system events.

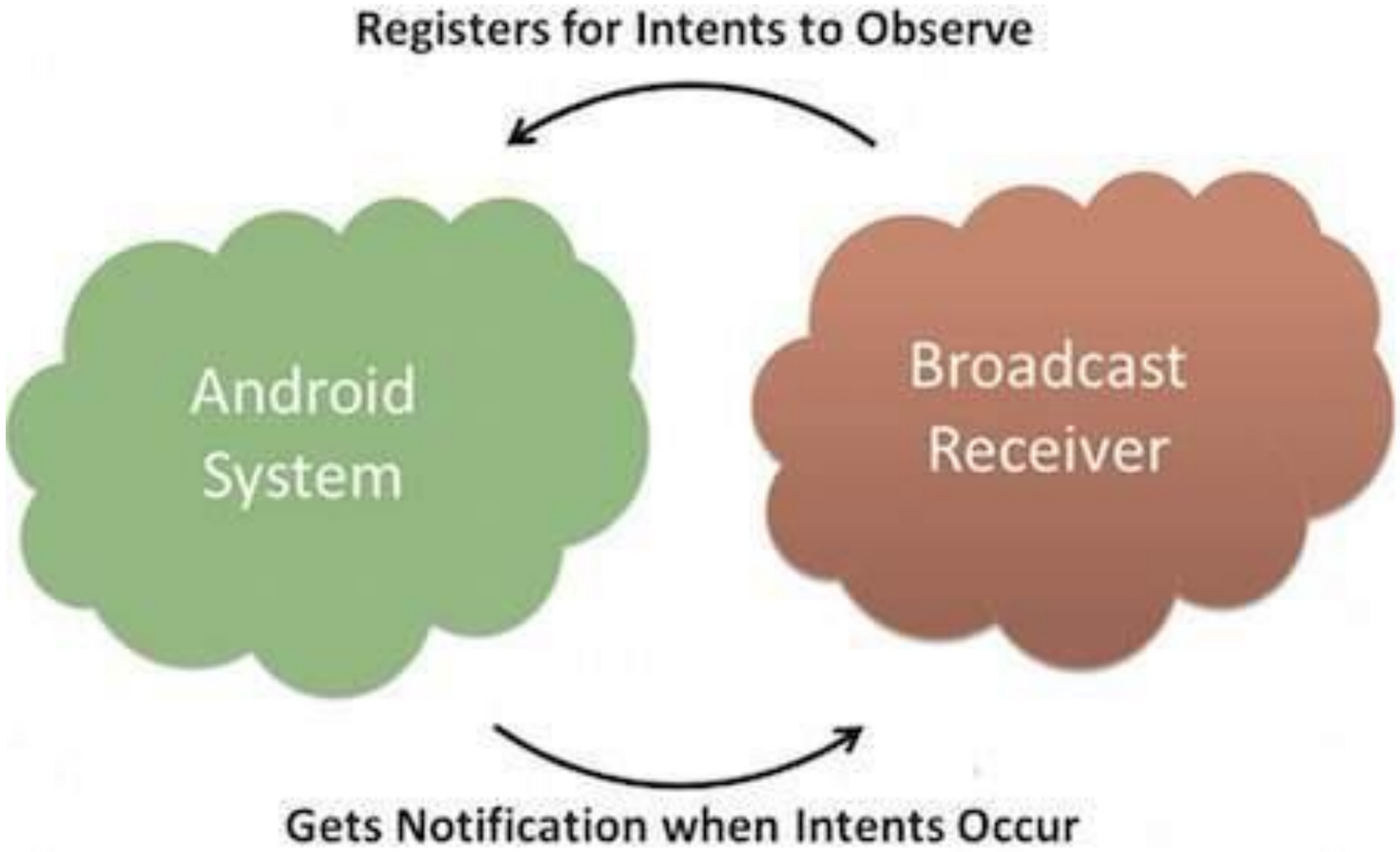
# Android - Broadcast Receivers

| Sr.No | Event Constant & Description |
|---|---|
| 1 | android.intent.action.BATTERY_CHANGED<br>Sticky broadcast containing the charging state, level, and other information about the battery. |
| 2 | android.intent.action.BATTERY_LOW<br>Indicates low battery condition on the device. |
| 3 | android.intent.action.BATTERY_OKAY<br>Indicates the battery is now okay after being low. |
| 4 | android.intent.action.BOOT_COMPLETED<br>This is broadcast once, after the system has finished booting. |
| 5 | android.intent.action.BUG_REPORT<br>Show activity for reporting a bug. |

# Android - Broadcast Receivers

| Sr.No | Event Constant & Description |
|---|---|
| 6 | android.intent.action.CALL<br>Perform a call to someone specified by the data. |
| 7 | android.intent.action.CALL_BUTTON<br>The user pressed the "call" button to go to the dialer or other appropriate UI for placing a call. |
| 8 | android.intent.action.DATE_CHANGED<br>The date has changed. |
| 9 | android.intent.action.REBOOT<br>Have the device reboot. |

# Android - Broadcast Receivers



# Android - Broadcast Receivers

**Broadcasting Custom Intents**

If you want your application itself should generate and send custom intents then you will have to create and send those intents by using the *sendBroadcast()* method inside your activity class.

If you use the *sendStickyBroadcast(Intent)* method, the Intent is **sticky**, meaning the *Intent* you are sending stays around after the broadcast is complete.

```
public void broadcastIntent(View view) {
   Intent intent = new Intent();
   intent.setAction("com.tutorialspoint.CUSTOM_INTENT");
   sendBroadcast(intent);
}
```

# Android - Broadcast Receivers

This intent *com.tutorialspoint.CUSTOM_INTENT* can also be registered in similar way as we have regsitered system generated intent.

```
<application
   android:icon="@drawable/ic_launcher"
   android:label="@string/app_name"
   android:theme="@style/AppTheme" >
   <receiver android:name="MyReceiver">

      <intent-filter>
         <action android:name="com.tutorialspoint.CUSTOM_INTENT">
         </action>
      </intent-filter>

   </receiver>
</application>
```

# Telephony.Sms

public static final class Telephony.Sms
extends Object implements BaseColumns, Telephony.TextBasedSmsColumns
java.lang.Object

↳ android.provider.Telephony.Sms

# Telephony.Sms

| Nested classes | |
|---|---|
| class | Telephony.Sms.Conversations<br>Contains all sent text-based SMS messages in the SMS app. |
| class | Telephony.Sms.Draft<br>Contains all sent text-based SMS messages in the SMS app. |
| class | Telephony.Sms.Inbox<br>Contains all text-based SMS messages in the SMS app inbox. |
| class | Telephony.Sms.Intents<br>Contains constants for SMS related Intents that are broadcast. |
| class | Telephony.Sms.Outbox<br>Contains all pending outgoing text-based SMS messages. |
| class | Telephony.Sms.Sent<br>Contains all sent text-based SMS messages in the SMS app. |

# Native Data Handling

What is native code for Android devices, and what is the NDK?

- Android apps run within the Dalvik virtual machine, which interprets device-agnostic, cross-platform commands into instructions for the specific device that it is running on.
- The speed and memory overhead is a worthwhile tradeoff. In some cases, developers need the absolute fastest performance possible.
- The NDK allows embedding C and C++ components within Android apps, allowing the most performance-intensive pieces to be as close to the hardware as possible.
- This comes at a cost, though — using native code complicates development.
- There are more tools to use and infrastructure to set up.
- Also, some details that were handled by the Dalvik virtual machine must now be handled by the developer.
- For these reasons, native code should be used only when necessary.

# Native Data Handling

When native code is needed?

- There are times that using native code can be advantageous, such as processing data or computing physics and graphics for games.
- Access to existing native libraries, as well as high-performance code, can also be good reasons.

Uses for the native code

- Game engine developers often dive right in to native code.
- The limited speed and memory of mobile devices means native code may be necessary to squeeze every bit of potential out for them

# Native Data Handling

- The native-activity sample resides under the NDK installation root, in samples/native-activity.
- It is a very simple example of a purely native application, with no Java source code.
- In the absence of any Java source, the Java compiler still creates an executable stub for the virtual machine to run.

```
#include <EGL/egl.h>
#include <GLES/gl.h>
#include <android/sensor.h>
#include <android/log.h>
#include <android_native_app_glue>
```

# Native Data Handling

- Create a new Native Activity project
- In this tutorial, we'll first create a new Android Native Activity project and then build and run the default app in the Visual Studio Emulator for Android.

To create a new project

- 1. Open Visual Studio. On the menu bar, choose File, New, Project.
- 2. In the New Project dialog box, under Templates, choose Visual C++, Cross Platform, and then choose the Native-Activity Application (Android) template.
- 3. Give the app a name like MyAndroidApp, and then choose OK.
- Visual Studio creates the new solution and opens Solution Explorer.

# Native Data Handling

- The new Android Native Activity app solution includes two projects:
- MyAndroidApp.NativeActivity contains the references and glue code for your app to run as a Native Activity on Android.
- The implementation of the entry points from the glue code are in main.cpp.
- Precompiled headers are in pch.h.
- This Native Activity app project is compiled into a shared library .so file which is picked up by the Packaging project.
- MyAndroidApp.Packaging creates the .apk file for deployment on an Android device or emulator.
- This contains the resources and AndroidManifest.xml file where you set manifest properties.
- It also contains the build.xml file that controls the Ant build process.
- It's set as the startup project by default, so that it can be deployed and run directly from Visual Studio.

# Native Data Handling

**Build and run the default Android Native Activity app**

- Build and run the app generated by the template to verify your installation and setup.
- For this initial test, run the app on one of the device profiles installed by the Visual Studio Emulator for Android.
- If you prefer to test your app on another target, you can load the target emulator or connect the device to your computer.

# Native Data Handling

**To build and run the default Native Activity app**

1. If it is not already selected, choose x86 from the Solution Platforms dropdown list.

If the Solution Platforms list isn’t showing, choose Solution Platforms from the Add/Remove Buttons list, and then choose your platform.

2. On the menu bar, choose Build, Build SolutionThe Output window displays the output of the build process for the two projects in the solution.

3. Choose one of the VS Emulator Android Phone (x86) profiles as your deployment target.

If you have installed other emulators or connected an Android device, you can choose them in the deployment target dropdown list.

# Native Data Handling

**To build and run the default Native Activity app**

4. Press F5 to start debugging, or Shift+F5 to start without debugging.

Visual Studio starts the emulator, which takes a few seconds to load and deploy your code. Once your app has started, you can set breakpoints and use the debugger to step through code, examine locals, and watch values.

5. Press Shift + F5 to stop debugging.

The emulator is a separate process that continues to run. You can edit, compile, and deploy your code multiple times to the same emulator.

# On Device File I/O

| sr.no | part & description |
|---|---|
| 1 | **prefix**<br>This is always set to content:// |
| 2 | **authority**<br>This specifies the name of the content provider, for example contacts, browser etc. For third-party content providers, this could be the fully qualified name, such as com.tutorialspoint.statusprovider |
| 3 | **data_type**<br>This indicates the type of data that this particular provider provides. For example, if you are getting all the contacts from the Contacts content provider, then the data path would be people and URI would look like thiscontent://contacts/people |
| 4 | **id**<br>This specifies the specific record requested. For example, if you are looking for contact number 5 in the Contacts content provider then URI would look like this content://contacts/people/5. |

# SharedPreferences

**Saving Key-Value Sets**

1. Get a Handle to a SharedPreferences
2. Write to Shared Preferences
3. Read from Shared Preferences

Using Shared Preferences

If you have a relatively small collection of key-values that you'd like to save, you should use the SharedPreferences APIs. A SharedPreferences object points to a file containing key-value pairs and provides simple methods to read and write them. Each SharedPreferences file is managed by the framework and can be private or shared.

# Get a Handle to a SharedPreferences

create a new shared preference file or access an existing one by calling one of two methods:

- getSharedPreferences() — Use this if you need multiple shared preference files identified by name, which you specify with the first parameter. You can call this from any Context in your app.
- getPreferences() — Use this from an Activity if you need to use only one shared preference file for the activity. Because this retrieves a default shared preference file that belongs to the activity, you don't need to supply a name.

For example, the following code is executed inside a Fragment. It accesses the shared preferences file that's identified by the resource string R.string.preference_file_key and opens it using the private mode so the file is accessible by only your app.

```
Context context = getActivity();
SharedPreferences sharedPref = context.getSharedPreferences(
        getString(R.string.preference_file_key), Context.MODE_PRIVATE);
```

When naming your shared preference files, you should use a name that's uniquely identifiable to your app, such as "com.example.myapp.PREFERENCE_FILE_KEY"

# Write to Shared Preferences

- To write to a shared preferences file, create a SharedPreferences.Editor by calling edit() on your SharedPreferences.

- Pass the keys and values you want to write with methods such as putInt() and putString(). Then call commit() to save the changes. For example:

```
SharedPreferences sharedPref =
getActivity().getPreferences(Context.MODE_PRIVATE);
SharedPreferences.Editor editor = sharedPref.edit();
editor.putInt(getString(R.string.saved_high_score), newHighScore);
editor.commit();
```

# Read from Shared Preferences

- To retrieve values from a shared preferences file, call methods such as getInt() and getString(), providing the key for the value you want, and optionally a default value to return if the key isn't present. For example:

```
SharedPreferences sharedPref =
getActivity().getPreferences(Context.MODE_PRIVATE);
int defaultValue = getResources().getInteger(R.string.saved_high_score_default);
long highScore = sharedPref.getInt(getString(R.string.saved_high_score),
defaultValue);
```

# Mobile databases - SQLite

- SQLite is a opensource SQL database that stores data to a text file on a device. Android comes in with built in SQLite database implementation.

- SQLite supports all the relational database features. In order to access this database, you don't need to establish any kind of connections for it like JDBC,ODBC e.t.c

**Database - Package**

- The main package is android.database.sqlite that contains the classes to manage your own databases

# Mobile databases - SQLite

- Database – Creation

- In order to create a database you just need to call this method openOrCreateDatabase with your database name and mode as a parameter. It returns an instance of SQLite database which you have to receive in your own object.Its syntax is given below
- SQLiteDatabase mydatabase = openOrCreateDatabase("your database name",MODE_PRIVATE,null);
- Apart from this , there are other functions available in the database package , that does this job. They are listed below

# Mobile databases - SQLite

| Sr.No | Method & Description |
|---|---|
| 1 | **openDatabase(String path, SQLiteDatabase.CursorFactory factory, int flags, DatabaseErrorHandler errorHandler)**<br>This method only opens the existing database with the appropriate flag mode. The common flags mode could be OPEN_READWRITE OPEN_READONLY |
| 2 | **openDatabase(String path, SQLiteDatabase.CursorFactory factory, int flags)**<br>It is similar to the above method as it also opens the existing database but it does not define any handler to handle the errors of databases |
| 3 | **openOrCreateDatabase(String path, SQLiteDatabase.CursorFactory factory)**<br>It not only opens but create the database if it not exists. This method is equivalent to openDatabase method. |
| 4 | **openOrCreateDatabase(File file, SQLiteDatabase.CursorFactory factory)**<br>This method is similar to above method but it takes the File object as a path rather then a string. It is equivalent to file.getPath() |

# Database - Insertion

- We can create table or insert data into table using execSQL method defined in SQLiteDatabase class. Its syntax is given below

- mydatabase.execSQL("CREATE TABLE IF NOT EXISTS TutorialsPoint(Username VARCHAR,Password VARCHAR);");
  mydatabase.execSQL("INSERT INTO TutorialsPoint VALUES('admin','admin');");

- This will insert some values into our table in our database. Another method that also does the same job but take some additional parameter is given below

| Sr.No | Method & Description |
|---|---|
| 1 | **execSQL(String sql, Object[] bindArgs)**<br>This method not only insert data , but also used to update or modify already existing data in database using bind arguments |

# Database - Fetching

- We can retrieve anything from database using an object of the Cursor class. We will call a method of this class called rawQuery and it will return a resultset with the cursor pointing to the table. We can move the cursor forward and retrieve the data.

```
Cursor resultSet = mydatbase.rawQuery("Select * from TutorialsPoint",null);
resultSet.moveToFirst();
String username = resultSet.getString(0);
String password = resultSet.getString(1);
```

- There are other functions available in the Cursor class that allows us to effectively retrieve the data. That includes

# Database - Fetching

| Sr.No | Method & Description |
|---|---|
| 1 | **getColumnCount()**<br>This method return the total number of columns of the table. |
| 2 | **getColumnIndex(String columnName)**<br>This method returns the index number of a column by specifying the name of the column |
| 3 | **getColumnName(int columnIndex)**<br>This method returns the name of the column by specifying the index of the column |
| 4 | **getColumnNames()**<br>This method returns the array of all the column names of the table. |
| 5 | **getCount()**<br>This method returns the total number of rows in the cursor |
| 6 | **getPosition()**<br>This method returns the current position of the cursor in the table |
| 7 | **isClosed()**<br>This method returns true if the cursor is closed and return false otherwise |

# Database - Helper class

- For managing all the operations related to the database , an helper class has been given and is called SQLiteOpenHelper. It automatically manages the creation and update of the database. Its syntax is given below

```
public class DBHelper extends SQLiteOpenHelper {
   public DBHelper(){
      super(context,DATABASE_NAME,null,1);
   }
   public void onCreate(SQLiteDatabase db) {}
   public void onUpgrade(SQLiteDatabase database, int oldVersion, int newVersion) {}
}
```

**Example**

- Here is an example demonstrating the use of SQLite Database. It creates a basic contacts applications that allows insertion, deletion and modification of contacts.
- To experiment with this example, you need to run this on an actual device on which camera is supported.

# Database - Helper class

| Steps | Description |
|---|---|
| 1 | You will use Android studio to create an Android application under a package com.example.sairamkrishna.myapplication. |
| 2 | Modify src/MainActivity.java file to get references of all the XML components and populate the contacts on listView. |
| 3 | Create new src/DBHelper.java that will manage the database work |
| 4 | Create a new Activity as DisplayContact.java that will display the contact on the screen |
| 5 | Modify the res/layout/activity_main to add respective XML components |
| 6 | Modify the res/layout/activity_display_contact.xml to add respective XML components |
| 7 | Modify the res/values/string.xml to add necessary string components |
| 8 | Modify the res/menu/display_contact.xml to add necessary menu components |
| 9 | Create a new menu as res/menu/mainmenu.xml to add the insert contact option |
| 10 | Run the application and choose a running android device and install the application on it and verify the results. |

# Data Sharing across APPs

- OS apps can gain access to event information from the Calendar database on the device. Your app can fetch events within a date range, be notified when events change, and even directly create, edit, and synch events with a remote calendar. Access to the calendar is provided by the Event Kit framework.
- The UIActivityViewController class is a built-in view controller. You can use it to provide various built-in, standard services, such as a pasteboard for copying and cutting data from your app and pasting it in another one (and vice versa), for posting to social media sites from within your app, and for sending messages via e-mail and SMS.
- iOS provides a *keychain* as one of the facilities it offers. A keychain is an encrypted container that holds things like passwords and other information that needs to be secure on an app. Applications with the same app ID prefix can gain shared access to the elements of the keychain that they're supposed to jointly create and manage.
- Apple's iCloud is a cloud storage service where you (and your app) can store data and automatically have this data synchronized with all your devices. iCloud provides an application programming interface (API).

# Enterprise data access via Internet

- One of the biggest challenges to delivering enterprise mobile solutions is connecting to enterprise data. Customers and employees need access to their data to be productive. Since mobile solutions mean access from anywhere, how does an enterprise mobile app always have access to the data that makes it useful.

- The requirements for a solution should address these points:
- 1. Threat Protection
- 2. Edge Authentication
- 3. Configure (Not Code) Security
- 4. Single Sign-on User Authentication
- 5. Device Authentication
- 6. OAuth Support

# Enterprise data access via Intranet

- Applications can access a content provider indirectly with an Intent. The application does not call any of the methods of ContentResolver or ContentProvider. Instead, it sends an intent that starts an activity, which is often part of the provider's own application.
- The destination activity is in charge of retrieving and displaying the data in its UI. Depending on the action in the intent, the destination activity may also prompt the user to make modifications to the provider's data.
- An intent may also contain "extras" data that the destination activity displays in the UI; the user then has the option of changing this data before using it to modify the data in the provider.

- Implementing a content provider involves always the following steps:
- 1. Create a class that extends ContentProvider
- 2. Create a contract class
- 3. Create the UriMatcher definition
- 4. Implement the onCreate() method
- 5. Implement the getType() method
- 6. Implement the CRUD methods