CHAPTER – 5

ADVANCE TOPICS

User Notifications:

Introduction:

A **notification** is a message we can display to the user outside of our application's normal UI. When we tell the system to issue a notification, it first appears as an icon in the notification area. To see the details of the notification, the user opens the notification drawer. Both the notification area and the notification drawer are system controlled areas that the user can view at any time.

Android **Toast** class provides a handy way to show users alerts but problem is that these alerts are not persistent which means alert flashes on the screen for a few seconds and then disappears. To see the details of the notification, we will have to select the icon which will display notification drawer having detail about the notification. While working with emulator with virtual device, we will have to click and drag down the status bar to expand it which will give us detail.

Create and Send Notifications:

Step 1: Create Notification Builder:

As a first step is to create a notification builder using *NotificationCompat.Builder.build()*. We will use Notification Builder to set various Notification properties like small and large icons, title, priority etc.

Example:

- NotificationCompat.Builder mBuilder = new NotificationCompat.Builder(this)

Step 2: Setting Notification Properties:

Once we have **Builder** object, we can set its Notification properties using Builder object as per our requirement. But this is mandatory to set at least following:

- A small icon, set by **setSmallIcon()**
- A title, set by **setContentTitle()**
- Detail text, set by **setContentText()**

Example:

- mBuilder.setSmallIcon(R.drawable.notification_icon);
- mBuilder.setContentTitle("Notification Alert, Click Me!");
- mBuilder.setContentText("Hi, This is Android Notification Detail!");

Step 3: Attach Actions:

This is an optional part and required if we want to attach an action with the notification. An action allows users to go directly from the notification to an **Activity** in our application, where they can look at one or more events or do further work. The action is defined by a **PendingIntent** containing an **Intent** that starts an Activity in our application. To associate the **PendingIntent** with a gesture, call the appropriate method of ***NotificationCompat.Builder***.

For example, if we want to start Activity when the user clicks the notification text in the notification drawer, we add the **PendingIntent** by calling **setContentIntent()**.

A **PendingIntent** object helps us to perform an action on our applications behalf, often at a later time, without caring of whether or not our application is running. We take help of stack builder object which will contain an artificial back stack for the started Activity. This ensures that navigating backward from the Activity leads out of our application to the Home screen.

Example:

```
Intent resultIntent = new Intent(this, ResultActivity.class);
TaskStackBuilder stackBuilder = TaskStackBuilder.create(this);
stackBuilder.addParentStack(ResultActivity.class);
// Adds the Intent that starts the Activity to the top of the stack
stackBuilder.addNextIntent(resultIntent);
PendingIntent resultPendingIntent =
        stackBuilder.getPendingIntent(0, PendingIntent.FLAG_UPDATE_CURRENT);
mBuilder.setContentIntent(resultPendingIntent);
```

Step 4: Issue the Notification:

Finally, we pass the Notification object to the system by calling *NotificationManager.notify()* to send our notification. We need to Make sure to call **NotificationCompat.Builder.build()** method on builder object before notifying it. This method combines all of the options that have been set and return a new **Notification** object.

Example:

```
NotificationManager mNotificationManager =
        (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
// notificationID allows you to update the notification later on.
mNotificationManager.notify(notificationID, mBuilder.build());
```

Broadcast Receivers Class:

Introduction:

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 which will intercept this communication and will initiate appropriate action.

Creating Broadcast Receiver:

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

Example:

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

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. Now whenever our Android device gets booted, it will be intercepted by BroadcastReceiver *MyReceiver* and implemented logic inside *onReceive()* will be executed.

Example:

```
<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>
```

Broadcasting Custom Intents:

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

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

Example:

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

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

Manifest File:

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

System Generated Events:

- **android.intent.action.BATTERY_CHANGED:** Sticky broadcast containing the charging state, level, and other information about the battery.
- **android.intent.action.BATTERY_LOW:** Indicates low battery condition on the device.
- **android.intent.action.BATTERY_OKAY:** Indicates the battery is now okay after being low.
- **android.intent.action.BOOT_COMPLETED:** This is broadcast once, after the system has finished booting.
- **android.intent.action.BUG_REPORT:** Show activity for reporting a bug.
- **android.intent.action.CALL:** Perform a call to someone specified by the data.
- **android.intent.action.CALL_BUTTON:** The user pressed the "call" button to go to the dialer or other appropriate UI for placing a call.
- **android.intent.action.DATE_CHANGED:** The date has changed.
- **android.intent.action.REBOOT:** Have the device reboot.

Threads, AsyncTask and Handlers:

Threads:

A *thread* is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority. Each thread may or may not also be marked as a daemon. When code running in some thread creates a new Thread object, the new thread has its priority initially set equal to the priority of the creating thread, and is a daemon thread if and only if the creating thread is a daemon.

When a Java Virtual Machine starts up, there is usually a single non-daemon thread (which typically calls the method named main of some designated class). The Java Virtual Machine continues to execute threads until either of the following occurs:

- The exit method of class Runtime has been called and the security manager has permitted the exit operation to take place.
- All threads that are not daemon threads have died, either by returning from the call to the run method or by throwing an exception that propagates beyond the run method.

There are two ways to create a new thread of execution. One is to declare a class to be a subclass of Thread. This subclass should override the run method of class Thread. An instance of the subclass can then be allocated and started. For example, a thread that computes prime larger than a stated value could be written as follows:

```
class PrimeThread extends Thread {
      long minPrime;
      PrimeThread(long minPrime) {
        this.minPrime = minPrime;
      }
      public void run() {
        // compute primes larger than minPrime
        . . .
      }
   }
```

The following code would then create a thread and start it running:

```
PrimeThread p = new PrimeThread(143);
p.start();
```

The other way to create a thread is to declare a class that implements the Runnable interface. That class then implements the run method. An instance of the class can then be allocated, passed as an argument when creating Thread, and started. The same example in this other style looks like the following:

```
class PrimeRun implements Runnable {
   long minPrime;
   PrimeRun(long minPrime) {
      this.minPrime = minPrime;
   }
   public void run() {
      // compute primes larger than minPrime
      . . .
   }
}
```

The following code would then create a thread and start it running:

```
PrimeRun p = new PrimeRun(143);
new Thread(p).start();
```

Every thread has a name for identification purposes. More than one thread may have the same name. If a name is not specified when a thread is created, a new name is generated for it. Unless otherwise noted, passing a null argument to a constructor or method in this class will cause a NullPointerException to be thrown.

AsyncTask:

AsyncTask enables proper and easy use of the UI thread. This class allows to perform background operations and publish results on the UI thread without having to manipulate threads and/or handlers.

AsyncTask is designed to be a helper class around Thread and Handler and does not constitute a generic threading framework. AsyncTasks should ideally be used for short operations (a few seconds at the most). If we need to keep threads running for long periods of time, it is highly recommended to use the various API's provided by the jave.util.concurrent package such as Executor, ThreadPoolExecutor and FutureTask.

An asynchronous task is defined by a computation that runs on a background thread and whose result is published on the UI thread. An asynchronous task is defined by 3 generic types, called Params, Progress and Result and 4 stpes, called onPreExecute, doInBackground, onProgressUpdate and onPostExecute.

AsyncTask's Generic Types:

The three types used by an asynchronous task are the following:

1. **Params:** The type of the parameters sent to the task upon execution.
2. **Progress:** The type of the progress units published during the background computation.
3. **Result:** The type of the result of the background computation

The 4 Steps:

When an asynchronous task is executed, the task goes through 4 steps:

1. **onPreExecute():**

It is invoked on the UI thread before the task is executed. This step is normally used to setup the task, for instance by showing a progress bar in the user interface.

2. **doInBackground(Params...):**

It is invoked on the background thread immediately after onPreExecute() finishes executing. This step is used to perform background computation that can take a long time. The parameters of the asynchronous task are passed to this step. The result of the computation must be returned by this step and will be passed back to the last step. This step can also use publishProgress(Progress…) to publish one or more units of progress. These values are published on the UI thread, in the onProgressUpdate(Progress…) step.

3. **onProgressUpdate(Progress...):**

It is invoked on the UI thread after a call to publishProgress(Progress…). The timing of the execution is undefined. This method is used to display any form of progress in the user interface while the background computation is still executing. For instance, it can be used to animate a progress bar or show logs in a text field.

4. **onPostExecute(Result):**

It is invoked on the UI thread after the background computation finishes. The result of the background computation is passed to this step as a parameter.

Handler:

Handler is part of the Android system's framework for managing threads. A Handler object receives messages and runs code to handle the messages. A Handler allows us to send and process Message and Runnable objects associated with a thread's MessageQueue.

A Handler object registers itself with the thread in which it is created. It provides a channel to send data to this thread. For example, if we create a new Handler instance in the onCreate() method of our activity, it can be used to post data to the main thread. The data which can be posted via the Handler class can be an instance of the Message or the Runnable class. A Handler is particular useful if you have want to post multiple times data to the main thread.

There are two main uses for a Handler:

- to schedule messages and runnables to be executed as some point in the future
- to enqueue an action to be performed on a different thread than our own

Scheduling messages is accomplished with the following methods:

- post(Runnable)
- postAtTime(Runnable, long)
- postDelayed(Runnable, long),
- sendEmptyMessage(int)
- sendMessage(Message),

- sendMessageAtTime(Message, long)
- sendMessageDelayed(Message, long)

The *post* versions allow us to enqueue Runnable objects to be called by the message queue when they are received;

The *sendMessage* versions allow us to enqueue a Message object containing a bundle of data that will be processed by the Handler's handleMessage(Message) method (requiring that you implement a subclass of Handler).

When posting or sending to a Handler, we can either allow the item to be processed as soon as the message queue is ready to do so, or specify a delay before it gets processed or absolute time for it to be processed. The latter two allow us to implement timeouts, ticks, and other timing-based behavior.

Example

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="vertical" >
<ProgressBar
      android:id="@+id/progressBar1"style="?android:attr/progressBarStyleHorizontal"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:indeterminate="false"
      android:max="10"
      android:padding="4dip" >
   </ProgressBar>
<TextView
      android:id="@+id/textView1"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="" >
    </TextView>
  <Button
      android:id="@+id/button1"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:onClick="startProgress"
      android:text="Start Progress" >
  </Button>
</LinearLayout>

  public void startProgress(View view) {
      // do something long
      Runnable runnable = new Runnable() {
         @Override
           public void run() {
```

```
            for (int i = 0; i <= 10; i++) {
               final int value = i;
                doFakeWork();
               progress.post(new Runnable() {
              @Override
                   public void run() {
                      text.setText("Updating");
                      progress.setProgress(value);
                   }
               });
             }
          }
       };
            new Thread(runnable).start();
   }
   // Simulating something timeconsuming
   private void doFakeWork() {
      SystemClock.sleep(5000);e.printStackTrace();
   }
}
```

Alarms:

The android.app.AlarmManager class provides access to the system alarm services. They allow us to schedule our application to be run at some point in the future. When an alarm goes off, the Intent that had been registered for it is broadcast by the system, automatically starting the target application if it is not already running.

Registered alarms are retained while the device is asleep (and can optionally wake the device up if they go off during that time), but will be cleared if it is turned off and rebooted.

The Alarm Manager holds a CPU wake lock as long as the alarm receiver's onReceive() method is executing. This guarantees that the phone will not sleep until we have finished handling the broadcast. Once onReceive() returns, the Alarm Manager releases this wake lock. This means that the phone will in some cases sleep as soon as our onReceive() method completes. If our alarm receiver called Context.startService(), it is possible that the phone will sleep before the requested service is launched.

To prevent this, our BroadcastReceiver and Service will need to implement a separate wake lock policy to ensure that the phone continues running until the service becomes available.

Alarm Types:

- **RTC_Wakeup:** Wakes device, fires Intent at the specified clock time
- **RTC:** Doesn't wake device, fires Intent at the specified clock time
- **ELAPSED_REALTIME:** Doesn't Wake device, fires Intent at the specified time (interpreted relative to time since last boot)

- **ELAPSED_REALTIME_WAKEUP:** Wakes device, fires Intent at the specified time (interpreted relative to time since last boot)

Networking:

Android lets our application connect to the internet or any other local network and allows us to perform network operations. A device can have various types of network connections.

Checking Network Connection:

Before we perform any network operations, we must first check that we are connected to that network or internet etc. For this android provides **ConnectivityManager** class. We need to instantiate an object of this class by calling **getSystemService()** method. Its syntax is given below:

```
ConnectivityManager check = (ConnectivityManager)
this.context.getSystemService(Context.CONNECTIVITY_SERVICE);
```

Once we instantiate the object of ConnectivityManager class, we can use **getAllNetworkInfo** method to get the information of all the networks. This method returns an array of **NetworkInfo**. So we have to receive it like this.

```
NetworkInfo[] info = check.getAllNetworkInfo();
```

The last thing we need to do is to check **Connected State** of the network. Its syntax is given below:

```
for (int i = 0; i<info.length; i++){
    if (info[i].getState() == NetworkInfo.State.CONNECTED){
        Toast.makeText(context, "Internet is connected
        Toast.LENGTH_SHORT).show();
    }
}
```

A part from this connected states, there are other states a network can achieve. They are listed below:

- Connecting
- Disconnected
- Disconnecting
- Suspended
- Unknown

Performing Network Operations

After checking that we are connected to the internet, we can perform any network operation. Here we are fetching the html of a website from a url.

Android provides **HttpURLConnection** and URL class to handle these operations. We need to instantiate an object of URL class by providing the link of website. Its syntax is as follows:

```
String link = "http://www.google.com";
URL url = new URL(link);
```

After that we need to call **openConnection** method of url class and receive it in a **HttpURLConnection** object. After that we need to call the **connect()** method of **HttpURLConnection** class.

```
HttpURLConnection conn = (HttpURLConnection) url.openConnection();
conn.connect();
```

And the last thing we need to do is to fetch the HTML from the website. For this we will use **InputStream** and **BufferedReader** class. Its syntax is given below:

```
InputStream is = conn.getInputStream();
BufferedReader reader =new BufferedReader(new InputStreamReader(is, "UTF-8"));
String webPage = "",data="";
while ((data = reader.readLine()) != null){
webPage += data + "\n";
}
```

Apart from this connect method, there are other methods available in HttpURLConnection class. They are listed below:

- **disconnect():**

This method releases this connection so that its resources may be either reused or closed.

- **getRequestMethod():**

This method returns the request method which will be used to make the request to the remote HTTP server.

- **getResponseCode():**

This method returns response code returned by the remote HTTP server

- **setRequestMethod(String method):**

This method Sets the request command which will be sent to the remote HTTP server

- **usingProxy():**

This method returns whether this connection uses a proxy server or not