UNIT II

CHAPTER 5

Triggering & Scheduling

Syllabus

Notifications, Alarm managers, Transferring data efficiently

Syllabus Topic : Notifications

5.1 Notifications

- A notification is a message your app displays to the user outside your application's normal UI. When you tell the system to issue a notification, the notification first appears to the user as an icon in the *notification area*, on the left side of the status bar.

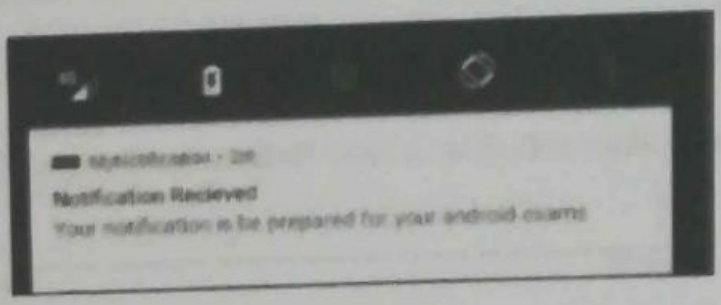


Fig. 5.1.1

- To see the details of the notification, the user opens the *notification drawer*, or views the notification on the lock screen if the device is locked. The notification area, the lock screen, and the notification drawer are system-controlled areas that the user can view at any time.
- The screenshot shows an "open" notification drawer. The status bar isn't visible, because the notification drawer is open.

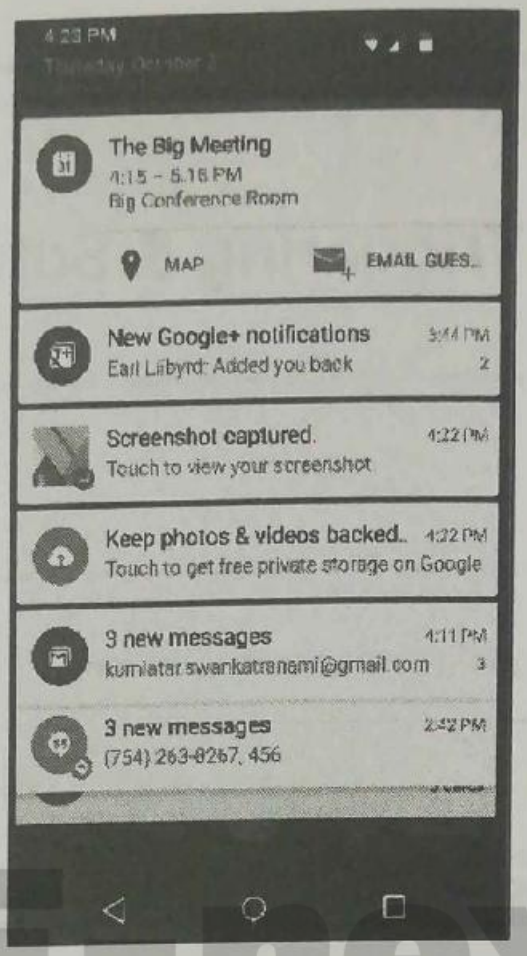


Fig. 5.1.2

5.1.1 Creating Notifications

- You create a notification using the NotificationCompat.Builder class. NotificationCompat for the best backward compatibility. The builder classes simplify the creation of complex objects.
- To create a NotificationCompat.Builder, pass the application context to the constructor:
- NotificationCompat.Builder mBuilder = new NotificationCompat.Builder(this);

```
NotificationCompat.Builder mBuilder = new NotificationCompat.Builder(this);
        |                                          |
      Object                                   Constructor
```

Fig. 5.1.3

5.1.2 Setting Notification Components

- When using NotificationCompat.Builder, you must assign a small icon, text for a title, and the notification message. You should keep the notification message shorter than 40 characters and not repeat what's in the title.

```
NotificationCompat.Builder mBuilder = new NotificationCompat.Builder(this)
.setSmallIcon(R.drawable.notification_icon)
.setContentTitle("Exam start!")
.setContentText("timetable is gien on your email id please check it");
```

- Now need to set an Intent that determines what happens when the user clicks the notification.
- Usually this Intent results in your app launching an Activity.
- To make sure the system delivers the Intent even when your app isn't running when the user clicks the notification, wrap the Intent in a PendingIntent object, which allows the system to deliver the Intentregardless of the app state.
- To instantiate a PendingIntent, use one of the following methods, depending on how you want the contained Intent to be delivered:
- To launch an Activity when a user clicks on the notification, use PendingIntent.getActivity(), passing inanexplicit Intent for the Activity you want to launch. The getActivity() method corresponds to an Intent delivered using startActivity().
- For an Intent passed into startService() (example a service to download a file), use PendingIntent.getService().
- For a broadcast Intent delivered with sendBroadcast(), use PendingIntent.getBroadcast().
- Each of these PendingIntent methods take the following arguments:
- The application context.
- A request code, which is a constant integer ID for the PendingIntent.
- The Intent to be delivered.
- A PendingIntent flag that determines how the system handles multiple PendingIntent objects from the same application.

Example

```
Intent contentIntent = new Intent(this, ExampleActivity.class);
PendingIntent pendingContentIntent = PendingIntent.getActivity(this, 0, contentIntent, PendingIntent.FLAG_UPDATE_CURRENT); mBuilder.setContentIntent(pendingContentIntent);
```

Optional Components

- Notification actions
- Priorities
- Expanded layouts
- Ongoing notifications

5.1.3 Notification Actions

A notification action is an action that the user can take on the notification.

- The action is made available via an action button on the notification. Like the Intent that determines what happens when the user clicks the notification, a notification action uses a PendingIntent to complete the action.
- This notification has two actions that the user can take, "Reply," or "Archive." Each has an icon.
- To add a notification action, use the addAction() method with the NotificationCompat.Builder object. Pass in the icon, the title string and the PendingIntent to trigger when the user taps the action. mBuilder.addAction(R.drawable.car, "Get Directions", mapPendingIntent);

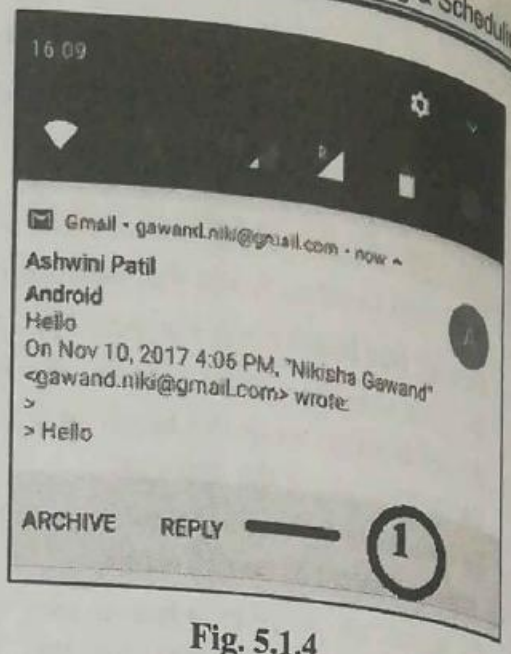


Fig. 5.1.4

5.1.4 Notification Priority

- Android allows you to assign a priority level to each notification to influence how the Android system will deliver it.
- Notifications have a priority between MIN (-2) and MAX (2) that corresponds to their importance. The Table 5.1.1 shows the available priority constants defined in the Notification class.

Table 5.1.1

Priority Constant	Use
PRIORITY_MAX	For critical and urgent notifications that alert the user to a condition that is time-critical or needs to be resolved before they can continue with a time-critical task.
PRIORITY_HIGH	Primarily for important communication, such as messages or chats.
PRIORITY_DEFAULT	For all notifications that don't fall into any of the other priorities described here.
PRIORITY_LOW	For information and events that are valuable or contextually relevant, but aren't urgent or time-critical.
PRIORITY_MIN	For nice-to-know background information. For example, weather or nearby places of interest.

- To change the priority of a notification, use the setPriority() method on the NotificationCompat.Builder object, passing in one of the above constants.

```
mBuilder.setPriority(Notification.PRIORITY_HIGH);
```

Peeking

Notifications with a priority of HIGH or MAX can peek, which means they slide briefly into view on the user's current screen, no matter what apps the user is using.

To create a notification that can peek

1. Set the priority to HIGH or MAX.
2. Set a sound or light pattern using the setDefaults() method on the builder, passing the DEFAULTS_ALL constant. This gives the notification a default sound, light pattern, and vibration.

```
NotificationCompat.Builder mBuilder =
new NotificationCompat.Builder(this)
setSmallIcon(R.drawable.notification_icon)
setContentTitle("My notification")
setContentText("Hello World!")
setPriority(PRIORITY_HIGH)
setDefaults(DEFAULTS_ALL);
```

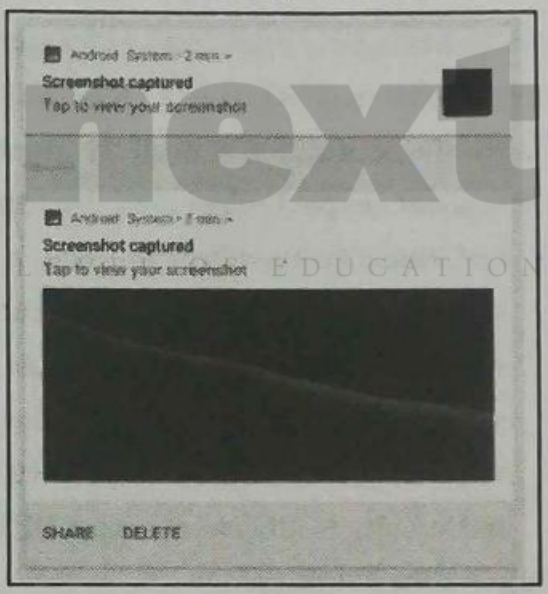
5.1.5 Expanded View Layouts

Notifications in the notification drawer appear in two main layouts,

normal view (which is the default)	*expanded view*

To create notifications that appear in an expanded layout, use one of these helper classes:

- Use NotificationCompat.BigTextStyle for large-format notifications that include a lot of text.
- Use NotificationCompat.InboxStyle for large-format notifications that include a list of up to five strings.
- Use Notification.MediaStyle for media playback notifications. There is currently no NotificationCompat version of this style, so it can only be used on devices with Android 4.1 or above.
- Use NotificationCompat.BigPictureStyle, shown in the screenshot below, for large-format notifications that include a large image attachment.



Example

here's how you'd set the BigPictureStyle on a notification:

```
NotificationCompat notif = new NotificationCompat.Builder(mContext)
    .setContentTitle("New photo from " + sender.toString())
    .setContentText(subject)
    .setSmallIcon(R.drawable.new_post)
    .setLargeIcon(aBitmap)
    .setStyle(new NotificationCompat.BigPictureStyle()
        .bigPicture(aBigBitmap)
        .setBigContentTitle("Large Notification Title"))
    .build();
```

☞ Ongoing notifications

- *Ongoing notifications* are notifications that can't be dismissed by the user. Your app must explicitly cancel them by calling cancel() or cancelAll(). Creating multiple ongoing notifications is a nuisance to your users since they are unable to cancel the notification. Use ongoing notifications sparingly.
- To make a notification ongoing, set setOngoing() to true. Use ongoing notifications to indicate background tasks that the user actively engages with (such as playing music) or tasks that occupy the device (such as file downloads, sync operations, and active network connections).

☞ Delivering notifications

Use the NotificationManager class to deliver notifications:

1. Call getSystemService(), passing in the NOTIFICATION_SERVICE constant, to create an instance of NotificationManager.
2. Call notify() to deliver the notification. In the notify() method, pass in these two values:

- A notification ID, which is used to update or cancel the notification.
- The Notification Compat object that you created using the Notification Compat. Builder object.

The following example creates a NotificationManager instance, then builds and delivers a notification:

```
mNotifyManager = (NotificationManager)
getSystemService(NOTIFICATION_SERVICE);
//Builds the notification with all the parameters
NotificationCompat.Builder notifyBuilder = new NotificationCompat.Builder(this)
    .setContentTitle(getString(R.string.notification_title))
    .setContentText(getString(R.string.notification_text))
    .setSmallIcon(R.drawable.ic_android)
    .setContentIntent(notificationPendingIntent)
    .setPriority(NotificationCompat.PRIORITY_HIGH)
    .setDefaults(NotificationCompat.DEFAULT_ALL);
//Delivers the notification
mNotifyManager.notify(NOTIFICATION_ID, notifyBuilder.build());
```

☞ Clearing notifications

Notifications remain visible until one of the following happens:

- If the notification can be cleared, it disappears when the user dismisses it individually or by using "Clear All."
- If you called setAutoCancel() when you created the notification, the notification disappears when the user clicks it.
- If you call cancel() for a specific notification ID, the notification disappears.
- If you call cancelAll(), all the notifications you've issued disappear.

Because ongoing notifications can't be dismissed by the user, your app must cancel them by calling cancel() or cancelAll().

Xml file

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res-auto"
xmlns:tools="http://schemas.android.com/tools"
android:layout_width="match_parent"
android:layout_height="match_parent"
tools:context="com.am.mumbai.mynotification.MainActivity">

<Button
  android:id="@+id/button"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:layout_alignParentBottom="true"
  android:layout_alignParentLeft="true"
  android:layout_alignParentStart="true"
  android:layout_marginBottom="17dp"
  android:layout_marginLeft="30dp"
  android:layout_marginStart="30dp"
  android:onClick="click"
  android:text="Click"
  android:textSize="24sp"
  android:textStyle="bold" />

<ImageView
  android:id="@+id/imageView"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  app:srcCompat="@drawable/noti"
  android:layout_marginBottom="241dp"

  android:layout_above="@+id/button"
  android:layout_alignParentLeft="true"
  android:layout_alignParentStart="true" />

<ImageView
  android:id="@+id/imageView3"

  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  app:srcCompat="@drawable/notify"
  android:layout_marginTop="86dp"
```

```
    android:layout_alignParentTop="true"
    android:layout_alignParentLeft="true"
    android:layout_alignParentStart="true" />


</RelativeLayout>
```

MainActivity.java file

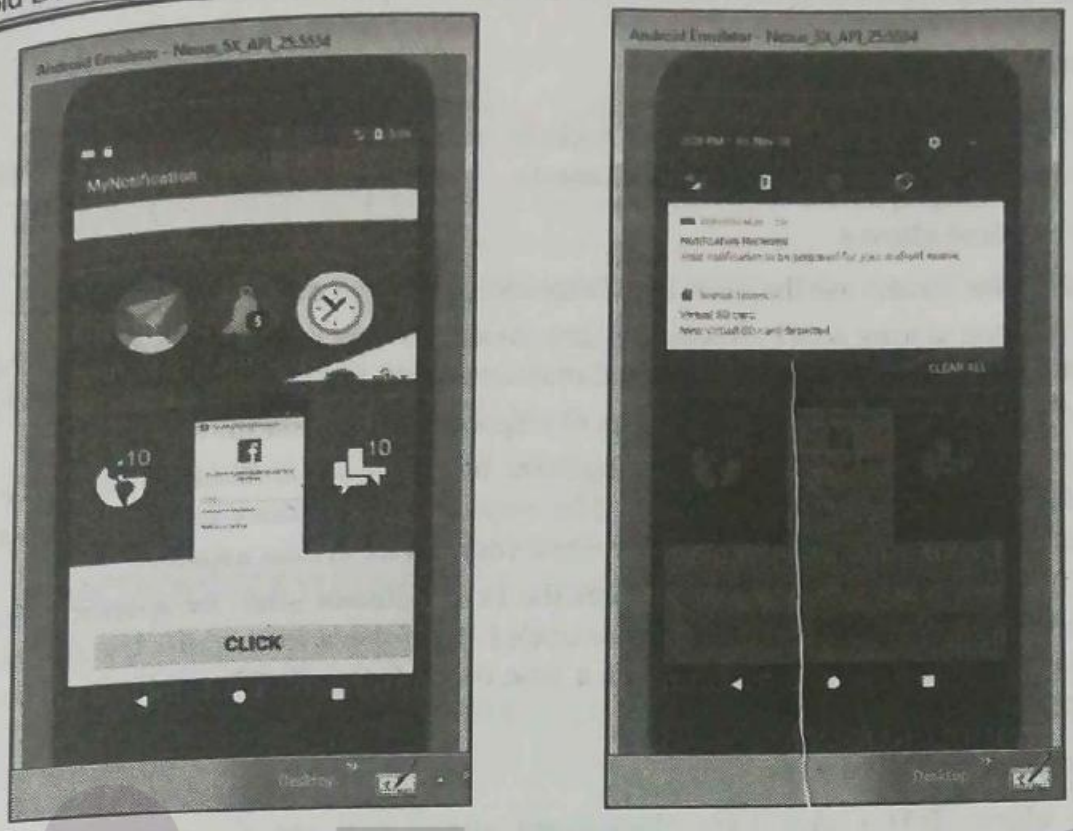
```
package com.am.mumbai.mynotification;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.app.NotificationManager;

import android.support.v7.app.NotificationCompat;
import android.view.View;

public class MainActivity extends AppCompatActivity {

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
  }
  public void click(View v)
  {
    NotificationCompat.Builder nb =new NotificationCompat.Builder(this);
    nb.setContentTitle("Notification Recieved ");
    nb.setContentText("Your notification is be prepared for your android exams ");
    nb.setSmallIcon(R.drawable.notification);
  NotificationManager nm=(NotificationManager)getSystemService(NOTIFICATION_SERVICE);
    nm.notify(0,nb.build());
    nb.setPriority(0);
  }
}
```



Syllabus Topic : Alarm Managers

5.2 Alarm Managers

- we know how to use broadcast receivers to make your app respond to system events even when your app isn't running. here we see how to use alarms to schedule tasks for specific times, whether or not your app is running at the time the alarm is set to go off.
- Alarms can either be single use or repeating. i.e. use a repeating alarm to schedule a download every day at the same time.
- To create alarms, use the AlarmManager class.

5.2.1 Alarms Characteristic in Android Alarms

- You send intents at set times or intervals. You can use alarms with broadcast receivers to start services and perform other operations.
- Alarms operate outside your app, so you can use them to trigger events or actions even when your app isn't running, and even if the device is asleep.
- When used correctly, alarms can help you minimize your app's resource requirements. For example, you can schedule operations without relying on timers or continuously running background services.

5.2.2 Alarm Types

There are two general types of alarms in Android: *elapsed real-time (ERT)*alarms and real-time clock *(RTC)*alarms, and both use PendingIntent objects.

1. **Elapsed real-time alarms**

- Elapsed real-time alarms use the time, in milliseconds, since the device was booted.
- Elapsed real-time alarms aren't affected by time zones, so they work well for alarms based on the passage of time. For example, use an elapsed real-time alarm for an alarm that fires every half hour.
- The AlarmManager class provides two types of elapsed real-time alarm:
- **ELAPSED_REALTIME :** Fires a PendingIntent based on the amount of time since the device was booted, but doesn't wake the device. The elapsed time includes any time during which the device was asleep. All repeating alarms fire when your device is next awake.
- **ELAPSED_REALTIME_WAKEUP:** Fires the PendingIntent after the specified length of time has elapsed since device boot, waking the device's CPU if the screen is off. Use this alarm instead of ELAPSED_REALTIME if your app has a time dependency, if it has a limited window during which to perform an operation.

2. **Real-time clock (RTC) alarms**

- Real-time clock (RTC) alarms are clock-based alarms that use Coordinated Universal Time (UTC).
- choose an RTC alarm in these types of situations:
- You need your alarm to fire at a particular time of day.
- The alarm time is dependent on current locale.
- Apps with clock-based alarms might not work well across locales, because they might fire at the wrong times. And if the user changes the device's time setting, it could cause unexpected behavior in your app.
- The AlarmManager class provides two types of RTC alarm:
- **RTC:** Fires the pending intent at the specified time but doesn't wake up the device. All repeating alarms fire when your device is next awake.
- RTC_WAKEUP: Fires the pending intent at the specified time, waking the device's CPU if the screen is off.

5.2.3 Scheduling an Alarm

The AlarmManager class gives you access to the Android system alarm services. AlarmManager lets you broadcast an Intent at a scheduled time, or after a specific interval.

1. Call getSystemService(ALARM_SERVICE) to get an instance of the AlarmManager class.
2. Use one of the set...() methods available in AlarmManager.

- This method you use depends on whether the alarm is elapsed real time, or RTC.
- All the AlarmManager.set...() methods include these two arguments:
  - A type **argument,** which is how you specify the alarm type:
  - ELAPSED_REALTIME or ELAPSED_REALTIME_WAKEUP.
  - RTC or RTC_WAKEUP.
  - A PendingIntent object, which is how you specify which task to perform at the given time.

Scheduling a single-use alarm

To schedule a single alarm, use one of the following methods on the AlarmManager instance:

set():	For devices running API 19+, this method schedules a single, inexactly timed alarm, meaning that the system shifts the alarm to minimize wakeups and battery use. For devices running lower API versions, this method schedules an exactly timed alarm
setWindow():	For devices running API 19+, use this method to set a window of time during which the alarm should be triggered.
setExact():	For devices running API 19+, this method triggers the alarm at an exact time. Use this method only for alarms that must be delivered at an exact time, for example an alarm clock that rings at a requested time. Exact alarms reduce the OS's ability to minimize battery use, so don't use them unnecessarily.

Example of using set() to schedule a single-use alarm

```
alarmMgr.set(AlarmManager.ELAPSED_REALTIME,
        SystemClock.elapsedRealtime() + 1000*300,
        alarmIntent);
```

- Here type is ELAPSED_REALTIME, which means that this is an elapsed real-time alarm. If the device is idle when the alarm is sent, the alarm does not wake the device.
- The alarm is sent 5 minutes (300,000 milliseconds) after the method returns.
- AlarmIntent is a PendingIntent broadcast that contains the action to perform when the alarm is sent.

Scheduling a repeating alarm

You can also use the AlarmManager to schedule repeating alarms, using one of the following methods :

setRepeating():	Prior to Android 4.4 (API Level 19), this method creates a repeating, exactly timed alarm. On devices running API 19 and higher, setRepeating() behaves exactly like setInexactRepeating().
setInexactRepeating():	This method creates a repeating, inexact alarm that allows for batching. When you use setInexactRepeating(), Android synchronizes repeating alarms from multiple apps and fires them at the same time. This reduces the total number of times the system must wake the device, thus reducing drain on the battery. As of API 19, all repeating alarms are inexact.

Example of using setInexactRepeating() to schedule a repeating alarm:

```
alarmMgr.setInexactRepeating(AlarmManager.RTC_WAKEUP,
        calendar.getTimeInMillis(),
        AlarmManager.INTERVAL_FIFTEEN_MINUTES,
        alarmIntent);
```

- Here The type is RTC_WAKEUP, which means that this is a clock-based alarm that wakes the device when the alarm is sent.

- The first occurrence of the alarm is sent immediately, because calendar.getTimeInMillis() returns the current time as UTC milliseconds.

  After the first occurrence, the alarm is sent approximately every 15 minutes.
- If the method were setRepeating() instead of setInexactRepeating(), and if the device were running an API version lower than 19, the alarm would be sent exactly every 15 minutes.
- Possible values for this argument are INTERVAL_DAY, INTERVAL_FIFTEEN_MINUTES, INTERVAL_HALF_DAY, INTERVAL_HALF_HOUR, INTERVAL_HOUR.
- AlarmIntent is the PendingIntent that contains the action to perform when the alarm is sent. This intent typically comes from IntentSender.getBroadcast().

5.2.4 Checking for an Existing Alarm

It's often useful to check whether an alarm is already set. For example, you may want to disable the ability to set another alarm if one already exists.

1. Create a PendingIntent that contains the same Intent used to set the alarm, but this time use the FLAG_NO_CREATE flag. With FLAG_NO_CREATE, a PendingIntent is only created if one with the same Intent already exists. Otherwise, the request returns null.
2. Check whether the PendingIntent is null:
   - If it's null, the alarm has not yet been set.
   - If it's not null, the PendingIntent already exists, meaning that the alarm has been set.

   For example, the following code returns true if the alarm contained in alarmIntent already exists:

```
boolean alarmExists =
 (PendingIntent.getBroadcast(this, 0,
    alarmIntent,
    PendingIntent.FLAG_NO_CREATE) != null);
```


5.2.5 Canceling an Alarm

To cancel an alarm, use cancel() and pass in the PendingIntent.

For example : alarmManager.cancel(alarmIntent);

xml file

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   tools:context="com.am.mumbai.alarm.MainActivity">

   <EditText
      android:id="@+id/time"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_alignParentLeft="true"
```

```
    android:layout_alignParentTop="true"
    android:layout_marginTop="28dp"
    android:ems="10"
    android:hint="Number of seconds"
    android:inputType="numberDecimal" />

  <Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignRight="@+id/time"
    android:layout_below="@+id/time"
    android:layout_marginRight="60dp"
    android:layout_marginTop="120dp"
    android:text="Start" />

  <ImageView
    android:id="@+id/imageView"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
    android:layout_marginBottom="12dp"
    app:srcCompat="@drawable/alarm" />
</RelativeLayout>
```

Create a new java class name as MyBroadcastReceiver

MyBroadcastReceiver.java file

```
package com.am.mumbai.alarm;
/**
 * Created by kbp on 11/11/2017.
 */
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.media.MediaPlayer;
import android.widget.Toast;
public class MyBroadcastReceiver extends BroadcastReceiver{
  MediaPlayer mp;
  @Override
  public void onReceive(Context context, Intent intent) {
    mp=MediaPlayer.create(context, R.raw.songname);
    mp.start();
    Toast.makeText(context, "Alarm....", Toast.LENGTH_LONG).show();
  }
}
```

MainActivity.java file

```
package com.am.mumbai.alarm;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.app.Activity;
import android.app.AlarmManager;
import android.app.PendingIntent;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Toast;
public class MainActivity extends AppCompatActivity {
Button b1;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    b1=(Button) findViewById(R.id.button1);
    b1.setOnClickListener(new OnClickListener() {

      @Override
      public void onClick(View v) {
        // TODO Auto-generated method stub
        startAlert();
      }
    });

  }  public void startAlert() {
    EditText text = (EditText) findViewById(R.id.time);
    int i = Integer.parseInt(text.getText().toString());
    Intent intent = new Intent(this, MyBroadcastReceiver.class);
    PendingIntent pendingIntent = PendingIntent.getBroadcast(
        this.getApplicationContext(), 234324243, intent, 0);
    AlarmManager alarmManager = (AlarmManager) getSystemService(ALARM_SERVICE);
    alarmManager.set(AlarmManager.RTC_WAKEUP, System.currentTimeMillis()
        + (i * 1000), pendingIntent);
    Toast.makeText(this, "Alarm set in " + i + " seconds",Toast.LENGTH_LONG).show();
  }

}
```

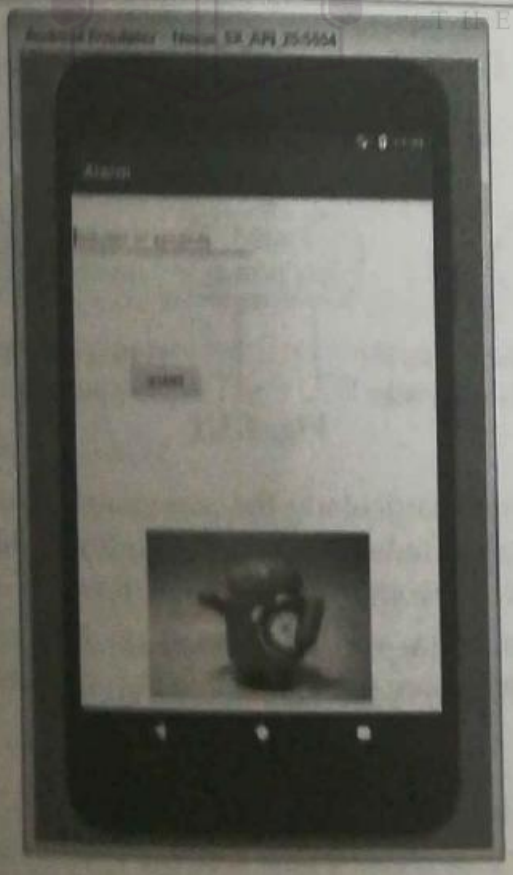
AndroidManifest.xml

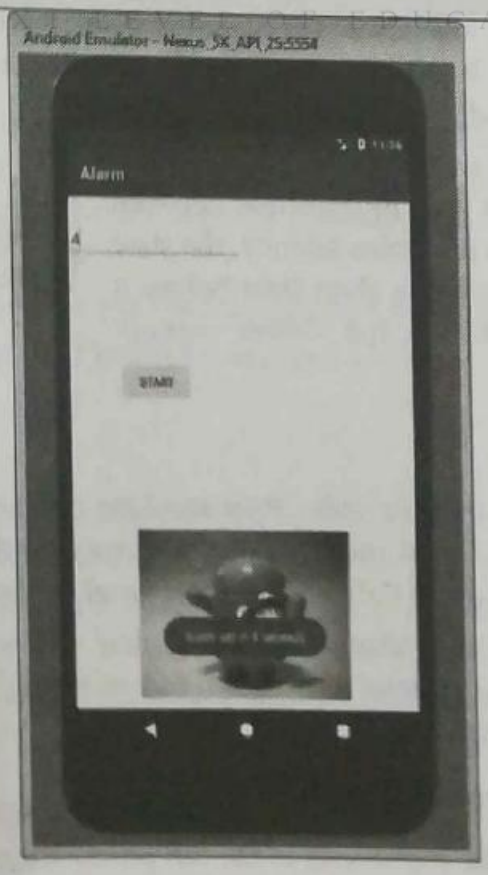
```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.an.numbra.alarm">

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="16" />

    <uses-permission android:name="android.permission.VIBRATE" />
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <receiver android:name="MyBroadcastReceiver" >
            </receiver>
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```





Syllabus Topic : Transferring Data Efficiently

5.3 Transferring Data Efficiently

- Transferring data is an essential part of most Android applications, but it can negatively affect battery life and increase data usage costs. Using the wireless radio to transfer data is potentially one of your app's most significant sources of battery drain.
- Users care about battery drain because they would rather use their mobile device without it connected to the charger. And users care about data usage, because every bit of data transferred can cost them money.
- Here we see how your app's networking activity affects the device's radio hardware so you can minimize the battery drain associated with network activity. also we see how to wait for the proper conditions to accomplish resource-intensive tasks.

5.3.1 Wireless Radio State

- A fully active wireless radio consumes significant power. To conserve power when not in use, the radio transitions between different energy states. However, there is a trade-off between conserving power and the time it takes to power up when needed.
- For a typical 3G network the radio has these three energy states:

1. **Full power:** It is used when a connection is active. and it allows the device to transfer data at its highest possible rate.
2. **Low power:** It is an intermediate state and uses about 50% less battery.
3. **Standby:** It is the minimal energy state, during this state no network connection is active or required.

- Whereas low state and standby state use much less battery, and introduce latency to network requests. It returning around 1.5 seconds takes for full power from the low state, and for moving from standby to full take over 2 seconds.
- Android uses a state machine to determine how to transition between states. To minimize latency, the state machine waits a short time before it transitions to the lower energy states.

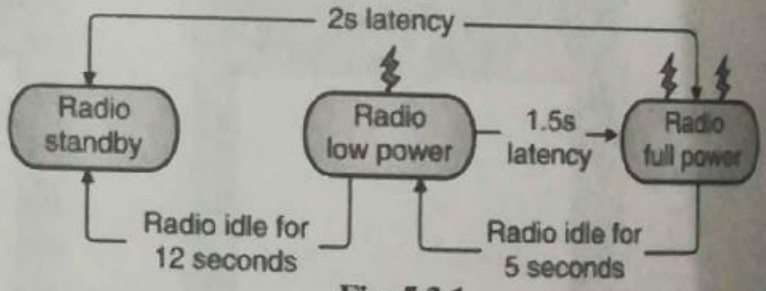


Fig. 5.3.1

- we know that the radio state machine on each device, particularly the associated transition delay ("tail time") and start up latency, it vary on the wireless radio technology employed like 2G, 3G, LTE, etc. and is defined and configured by the carrier network over which the device is operating.
- Example state machine for a typical 3G wireless radio, and it is based on data provided by AT&T. So general principles and resulting best practices are applicable for all wireless radio implementations.

Bundling network transfers

- For the radio transitions to the full power state every time create a new network connection.

In the case of the 3G radio state machine at full power for the duration of transfer, followed by 5 seconds of tail time, and followed by 12 seconds at the low energy state before turning off. So, for a typical 3G device, every data transfer session causes the radio to draw power for almost 20 seconds.

What this means in practice

We know that app that transfers unbundled data for 1 second every 18 seconds keeps the wireless radio always active ,hence by comparison, for the same app bundling transfers for 3 seconds of every minute keeps the radio low power state for an additional 12 seconds and high power state for only 8 seconds.

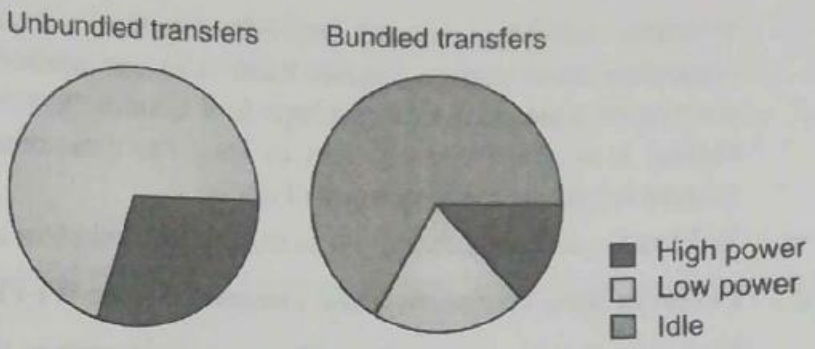


Fig. 5.3.2

5.3.2 Prefetching

To prefetch data means that your app takes a guess at what content or data the user will want next, and fetches it ahead of time.

For example, when the user looks at the first part of an article, a good guess is to prefetch the next part. Or, if a user is watching a video, fetching the next minutes of the video is also a good guess.

Prefetching data

- An effective way to reduce the number of independent data transfer sessions.
- Allows you to download all the data you are likely to need for a given time period in a single burst, over a single connection, at full capacity.
- This reduces the number of radio activations required to download the data.
- Hence as a result, you not only conserve battery life, but improve latency for the user, as lower the required bandwidth, and reduce download times.
- Prefetching has trade-offs. If you download too much or the wrong data, you might increase battery drain. And if you download at the wrong time, users may end up waiting. Optimizing prefetching data is an advanced topic not covered in this course, but the following guidelines cover common situations.
- Prefetch depends on the size of the data being downloaded and the likelihood of it being used.
- It's good practice to prefetch data such that you only need to initiate another download every 2 to 5 minutes, and on the order of 1 to 5 megabytes.

Prefetching example

- Many news apps attempt after a category has been selected to reduce bandwidth by downloading headlines only, only when the user wants to read them, and thumbnails just as they scroll into view.
- By using this approach, the radio is forced to remain active for the majority of a news-reading session as users scroll headlines, change categories, and read articles. but the constant switching between energy states results in significant latency when switching categories or reading articles.

Here's a better approach:

1. Prefetch a reasonable amount of data at startup, beginning with the first set of news headlines and thumbnails. This ensures a quick startup time.

2. Continue with the remaining headlines, the remaining thumbnails, and the article text for each article from the first set of headlines.

☞ **Monitor connectivity state**

1. Devices can network using different types of hardware:

- Wireless radios use varying amounts of battery depending on technology, and higher bandwidth consumes more energy. Higher bandwidth can prefetch downloading more data during the same amount of time. However, perhaps less intuitively, because the tail-time battery cost is relatively higher, it is also more efficient to keep the radio active for longer periods during each transfer session to reduce the frequency of updates.
- WiFi radio uses significantly less battery than wireless and offers greater bandwidth.

2. Perform data transfers when connected over Wi-Fi whenever possible.

You can use the ConnectivityManager to determine the active wireless radio and modify your prefetching routines depending on network type:

Example

```
ConnectivityManager cm =
  (ConnectivityManager)getSystemService(Context.CONNECTIVITY_SERVICE);
TelephonyManager tm =
  (TelephonyManager)getSystemService(Context.TELEPHONY_SERVICE);

NetworkInfo activeNetwork = cm.getActiveNetworkInfo();
int PrefetchCacheSize = DEFAULT_PREFETCH_CACHE;

switch (activeNetwork.getType()) {
  case (ConnectivityManager.TYPE_WIFI):
    PrefetchCacheSize = MAX_PREFETCH_CACHE; break;
  case (ConnectivityManager.TYPE_MOBILE): {
    switch (tm.getNetworkType()) {
      case (TelephonyManager.NETWORK_TYPE_LTE |
          TelephonyManager.NETWORK_TYPE_HSPAP):
        PrefetchCacheSize *= 4;
        break;
      case (TelephonyManager.NETWORK_TYPE_EDGE |
          TelephonyManager.NETWORK_TYPE_GPRS):
        PrefetchCacheSize /= 2;
        break;
      default: break;
```

```
    }
    break;
  }
default: break;
}
```

- The system sends out broadcast intents when the connectivity state changes, so you can listen for these changes using a BroadcastReceiver.

Monitor battery state

- To minimize battery drain, monitor the state of your battery and wait for specific conditions before initiating a battery-intensive operation.
- The BatteryManager broadcasts all battery and charging details in a broadcast Intent that includes the charging status.
- To check the current battery status, examine the broadcast intent:

```
IntentFilter ifilter = new IntentFilter(Intent.ACTION_BATTERY_CHANGED);
Intent batteryStatus = context.registerReceiver(null, ifilter);
// Are we charging / charged?
int status = batteryStatus.getIntExtra(BatteryManager.EXTRA_STATUS, -1);
boolean isCharging = status == BatteryManager.BATTERY_STATUS_CHARGING ||
                     status == BatteryManager.BATTERY_STATUS_FULL;
// How are we charging?
int chargePlug = batteryStatus.getIntExtra(BatteryManager.EXTRA_PLUGGED, -1);
boolean usbCharge = chargePlug == BatteryManager.BATTERY_PLUGGED_USB;
boolean acCharge = chargePlug == BatteryManager.BATTERY_PLUGGED_AC;
```

If you want to react to changes in the battery charging state, use a BroadcastReceiver registered for the battery status actions:

```
<receiver android:name=".PowerConnectionReceiver">
 <intent-filter>
  <action android:name="android.intent.action.ACTION_POWER_CONNECTED"/>
  <action android:name="android.intent.action.ACTION_POWER_DISCONNECTED"/>
 </intent-filter>
</receiver>
```

Broadcast intents are also delivered when the battery level changes in a significant way:

```
"android.intent.action.BATTERY_LOW"
"android.intent.action.BATTERY_OKAY"
```

5.3.3 JobScheduler

- Constantly monitoring the connectivity and battery status of the device can be a challenge, and it requires using components such as broadcast receivers, which can consume system resources even when your app isn't running.
- Because transferring data efficiently is such a common task, the Android SDK provides a class that makes this much easier: JobScheduler.

JobScheduler has three components:

- JobInfo uses the builder pattern to set the conditions for the task.
- JobService is a wrapper around the Service class where the task is actually completed.
- JobScheduler schedules and cancels tasks.
- JobScheduler is only available from API 21+. There is no backwards compatible version for prior API releases. If your app targets devices with earlier API levels, you might find the FirebaseJobDispatcher a useful alternative.

1. JobInfo

- Set the job conditions by constructing a JobInfo object using the JobInfo.Builder class. The JobInfo.Builder class is instantiated from a constructor that takes two arguments: a job ID (which can be used to cancel the job), and the ComponentName of the JobService that contains the task.
- Your JobInfo.Builder must set at least one, non-default condition for the job. For example:

```
JobScheduler scheduler = (JobScheduler) getSystemService(JOB_SCHEDULER_SERVICE);
ComponentName serviceName = new ComponentName(getPackageName(),
NotificationJobService.class.getName());
JobInfo.Builder builder = new JobInfo.Builder(JOB_ID, serviceName);
builder.setRequiredNetworkType(NETWORK_TYPE_UNMETERED);
JobInfo jobInfo = builder.build();
```

- The JobInfo.Builder class has many set() methods that allow you to determine the conditions of the task.
- Below is a list of available constraints with their respective set() methods and class constants:
  - Backoff/Retry policy: Determines when how the task should be rescheduled if it fails. Set this condition using the setBackoffCriteria() method, which takes two arguments: the initial time to wait after the task fails, and the backoff strategy. The backoff strategy argument can be one of two constants: BACKOFF_POLICY_LINEAR or BACKOFF_POLICY_EXPONENTIAL. This defaults to {30 seconds, Exponential}.
  - Minimum Latency: The minimum amount of time to wait before completing the task. Set this condition using the setMinimumLatency() method, which takes a single argument: the amount of time to wait in milliseconds.
  - Override Deadline: The maximum time to wait before running the task, even if other conditions aren't met. Set this condition using the setOverrideDeadline() method, which is the maximum time to wait in milliseconds.
  - Periodic: Repeats the task after a certain amount of time. Set this condition using the setPeriodic() method, passing in the repetition interval. This condition is mutually exclusive

with the minimum latency and override deadline conditions: setting setPeriodic() with one of them results in an error.

- Persisted: Sets whether the job is persisted across system reboots. For this condition to work, your app must hold the RECEIVE_BOOT_COMPLETED permission. Set this condition using the setPersisted()method, passing in a boolean that indicates whether or not to persist the task.
- Required Network Type: The kind of network type your job needs. If the network isn't necessary, you don't need to call this function, because the default is NETWORK_TYPE_NONE. Set this condition using the setRequiredNetworkType() method, passing in one of the following constants: NETWORK_TYPE_NONE, NETWORK_TYPE_ANY,NETWORK_TYPE_NOT_ROAMING, NETWORK_TYPE_UNMETERED.
- Required Charging State: Whether or not the device needs to be plugged in to run this job. Set this condition using the setRequiresCharging() method, passing in a boolean. The default is false.
- Requires Device Idle: Whether or not the device needs to be in idle mode to run this job. "Idle mode" means that the device isn't in use and hasn't been for some time, as loosely defined by the system. When the device is in idle mode, it's a good time to perform resource-heavy jobs. Set this condition using the setRequiresDeviceIdle() method, passing in a boolean. The default is false.

2. JobService

- Once the conditions for a task are met, the framework launches a subclass of JobService, which is where you implement the task itself. The JobService runs on the UI thread, so you need to offload blocking operations to a worker thread.
- Declare the JobService subclass in the Android Manifest, and include the BIND_JOB_SERVICE permission:

```
<service android:name="MyJobService"
        android:permission="android.permission.BIND_JOB_SERVICE" />
```

In your subclass of JobService, override two methods, onStartJob() and onStopJob().

onStartJob()

- The system calls onStartJob() and automatically passes in a JobParameters object, which the system creates with information about your job. If your task contains long-running operations, offload the work onto a separate thread.
- The onStartJob() method returns a boolean: true if your task has been offloaded to a separate thread (meaning it might not be completed yet) and false if there is no more work to be done.
- Use the jobFinished() method from any thread to tell the system that your task is complete.
- This method takes two parameters: the JobParameters object that contains information about the task, and a boolean that indicates whether the task needs to be rescheduled, according to the defined backoff policy.

onStopJob()

- The system calls onStopJob() if it determines that you must stop execution of your job even before you've call jobFinished(). This happens if the requirements that you specified when you scheduled the job are no longer met.

Examples:

- If you request WiFi with setRequiredNetworkType() but the user turns off WiFi while while your job is executing, the system calls onStopJob().
- If you specify setRequiresDeviceIdle() but the user starts interacting with the device while your job is executing, the system calls onStopJob().
- You're responsible for how your app behaves when it receives onStopJob(), so don't ignore it. This method returns a boolean, indicating whether you'd like to reschedule the job based on the defined backoff policy, or drop the task.

3. JobScheduler

- The final part of scheduling a task is to use the JobScheduler class to schedule the job. To obtain an instance of this class, call getSystemService(JOB_SCHEDULER_SERVICE).
- Then schedule a job using the schedule() method, passing in the JobInfo object you created with the JobInfo.Builder. For example: mScheduler.schedule(myJobInfo);
- The framework is intelligent about when you receive callbacks, and it attempts to batch and defer them as much as possible. Typically, if you don't specify a deadline on your job, the system can run it at any time, depending on the current state of the JobScheduler object's internal queue; however, it might be deferred as long as until the next time the device is connected to a power source.
- To cancel a job, call cancel(), passing in the job ID from the JobInfo.Builder object, or call cancelAll(). For example:
- mScheduler.cancelAll();

Review Questions

Q. 1 What is notifications? **(Refer section 5.1)**

Q. 2 How to set notifications component? **(Refer section 5.1.2)**

Q. 3 Explain Alarm types. **(Refer section 5.2.2)**

Q. 4 How to schedule alarm? **(Refer section 5.2.3)**

Q. 5 Explain the concept of prefetching. **(Refer section 5.3.2)**

Q. 6 Explain in detail JobScheduler. **(Refer section 5.3.3)**

❑❑❑