NOTIFICATIONS AND ALARMS

A notification is a message you can display to the user outside of your application's normal UI. When you 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.

Notifications are designed to tell users when apps that are inactive or running in the background have new information, including messages, upcoming events, or other relevant and timely data.

Notifications are displayed in three ways:

Alerts or

banners App

badges

Sounds or vibrations

Android Notification provides short, timely information about the action happened in the application, even it is not running. The notification displays the icon, title and some amount of the content text.

SET ANDROID NOTIFICATION PROPERTIES

The properties of Android notification are set using NotificationCompat.Builder object.

Some of the notification properties are mention below:

o setSmallIcon(): It sets the icon of notification.

o setContentTitle(): It is used to set the title of notification.

o setContentText(): It is used to set the text message.

o setAutoCancel(): It sets the cancelable property of notification.

o setPriority(): It sets the priority of notification.

Notifications are a way for your applications to alert users, without using an Activity.

Notifications are handled by the Notification Manger, and currently include the ability to:

• Create a new status bar icon.

• Display additional information (and launch an Intent) in the extended status bar window.

• Flash the lights/LEDs.

• Vibrate the phone.

• Sound audible alerts (ringtones, media store sounds).

Notifications are the preferred way for invisible application components (Broadcast Receivers, Services, and inactive Activities) to alert users that events have occurred that require attention.

• As a User Interface metaphor, Notifications are particularly well suited to mobile devices. It‘s likely that your users will have their phones with them at all times but quite unlikely that they will be paying attention to them, or your application, at any given time.

• Generally, users will have several applications open in the background, and they won‘t be paying attention to any of them. In this environment, it‘s important that your applications be able to alert users when specific events occur that require their attention.

• Notifications can be persisted through insistent repetition, or (more commonly) by using an icon on the status bar. Status bar icons can be updated regularly or expanded to show additional information using the expanded status bar window shown in Figure.

INTRODUCING THE NOTIFICATION MANAGER

The Notification Manager is a system Service used to handle Notifications. Get a reference to

it using the getSystemService method, as shown in the snippet below:

String svcName = Context.NOTIFICATION_SERVICE;

NotificationManager notificationManager;

notificationManager = (NotificationManager)getSystemService(svcName);

Using the Notification Manager, you can trigger new Notifications, modify existing ones, or

remove those that are no longer needed or wanted.

TRIGGERING NOTIFICATIONS

To fire a Notification, pass it in to the notify method on the Notification Manager along with an integer reference ID, as shown in the following snippet:

Int notificationRef =1;

notificationManager.notify(notificationRef,notification);

ADVANCED NOTIFICATION TECHNIQUES

1. MAKING SOUNDS

Using an audio alert to notify the user of a device event (like incoming calls) is a technique that predates the mobile, and has stood the test of time. Most native phone events from incoming calls to new messages and low battery are announced by an audible ringtone. Android lets you play any audio file on the phone as a Notification by assigning a location URI to the sound property, as shown in the snippet below:

notification.sound = ringURI;

2. IBRATING THE PHONE

You can use the phone‘s vibration function to execute a vibration pattern specific to your

Notification. Android lets you control the pattern of a vibration; you can use vibration to convey information as well as get the user‘s attention.

To set a vibration pattern, assign an array of longs to the Notification‘s vibrate property. Construct the array so that every alternate number is the length of time (in milliseconds) to vibrate or pause, respectively.

Before you can use vibration in your application, you need to be granted permission. Add a uses- To update a Notification that‘s already been fired, re-trigger, passing the same reference ID. You can pass in either the same Notification object or an entirely new one. As long as the ID values are the same, the new Notification will be used to replace the status icon and extended status window details.

You also use the reference ID to cancel Notifications by calling the cancel method on the Notification Manager, as shown below:

```
notificationManager.cancel(notificationRef);
```

Canceling a Notification removes its status bar icon and clears it from the extended status

window.

3. MAKING SOUNDS

Using an audio alert to notify the user of a device event (like incoming calls) is a technique that predates the mobile, and has stood the test of time. Most native phone events from incoming calls to new messages and low battery are announced by an audible ringtone. Android lets you play any audio file on the phone as a Notification by assigning a location URI to the sound property, as shown in the snippet below:

```
notification.sound = ringURI;
```

4. IBRATING THE PHONE

You can use the phone‘s vibration function to execute a vibration pattern specific to your Notification. Android lets you control the pattern of a vibration; you can use vibration to convey information as well as get the user‘s attention.

To set a vibration pattern, assign an array of longs to the Notification‘s vibrate property. Construct the array so that every alternate number is the length of time (in milliseconds) to vibrate or pause, respectively.

Before you can use vibration in your application, you need to be granted permission. Add a uses- permission to your application to request access to the device vibration using the following code snippet:

```
<uses-permission android:name=”android.permission.VIBRATE”/>
```

The following example shows how to modify a Notification to vibrate in a repeating pattern of 1 second on, 1 second off, for 5 seconds total.

```
long[] vibrate = new long[] { 1000, 1000, 1000, 1000, 1000};

notification.vibrate = vibrate;
```

5. FLASHING THE LIGHTS

Notifications also include properties to configure the color and flash frequency of the device‘s LED. The ledARGB property can be used to set the LED‘s color, while the ledOffMS and ledOnMS properties let you set the frequency and pattern of the flashing LED. You can turn the LED on by setting the ledOnMS property to 1 and the ledOffMS property to 0, or turn it off by setting both properties to 0. Once you have configured the LED settings, you must also add the FLAG_SHOW_LIGHTS flag to the Notification‘s flags property.

The following code snippet shows how to turn on the red device LED:

```
notification.ledARGB = Color.RED;

notification.ledOffMS = 0;

notification.ledOnMS = 1;

notification.flags = notification.flags | Notification.FLAG_SHOW_LIGHTS;
```

ALARMS

Alarms are an application independent way of firing Intents at predetermined times. Alarms are set outside the scope of your applications, so they can be used to trigger application events or actions even after your application has been closed. They can be particularly powerful in combination with Broadcast Receivers, allowing you to set Alarms that launch applications or perform actions without applications needing to be open and active until they‘re required.

Alarms in Android remain active while the device is in sleep mode and can optionally be set to wake the device; however, all Alarms are canceled whenever the device is rebooted. Alarm

operations are handled through the AlarmManager, a system Service accessed via getSystemService as shown below:

```
AlarmManager alarms

=(AlarmManager)getSystemService(Context.ALARM_SERVICE);
```

To create a new Alarm, use the set method and specify an alarm type, trigger time, and a Pending Intent to fire when the Alarm triggers. If the Alarm you set occurs in the past, it will be triggered immediately.

CHOOSE AN ALARM TYPE

One of the first considerations in using a repeating alarm is what its type should be. The following four alarm types are available:

a. RTC_WAKEUP — Wakes the device from sleep to fire the Pending Intent at the clock time specified.

b. RTC — Fires the Pending Intent at the time specified but does not wake the device.

c. ELAPSED_REALTIME — Fires the Pending Intent based on the amount of time elapsed since the device was booted but does not wake the device. The elapsed time includes any period of time the device was asleep.

d. ELAPSED_REALTIME_WAKEUP — Wakes the device from sleep and fires the Pending Intent after a specified length of time has passed since device boot.

• Elapsed real time uses the "time since system boot" as a reference, and real time clock uses UTC (wall clock) time. This means that elapsed real time is suited to setting an alarm based on the passage of time (for example, an alarm that fires every 30 seconds) since it isn't affected by time zone/locale. The real time clock type is better

• suited for alarms that are dependent on current locale.

• Both types have a "wakeup" version, which says to wake up the device's CPU if the screen is off. This ensures that the alarm will fire at the scheduled time. This is useful if your app has a time dependency for example, if it has a limited window to perform a particular operation. If you don't use the wakeup version of your alarm type, then all the repeating alarms will fire when your device is next awake.

• If you simply need your alarm to fire at a particular interval (for example, every half hour), use one of the elapsed real time types. In general, this is the better choice.

• If you need your alarm to fire at a particular time of day, then choose one of the clockbased real time clock types. This approach can have some drawbacks the app may not translate well to other locales, and if the user changes the device's time setting, it could cause unexpected behavior in your app. Using a real time clock alarm type also does not scale well, as discussed above.

SELECTING AND REPEATING ALARMS

• Repeating alarms work in the same way as the one-shot alarms but will trigger repeatedly at thespecified interval.

• Because alarms are set outside your Application lifecycle, they are perfect for scheduling regular updatesor data lookups so that they don‘t require a Service to be constantly running in the background.

• To set a repeating alarm, use the setRepeatingor setInexactRepeatingmethod on the

Alarm Manager. Both methods support an alarm type, an initial trigger time, and a Pending Intent to fire when the alarm triggers Use setRepeating when you need fine- grained control over the exact interval of your repeating alarm. The interval value passed in to this method lets you specify an exact interval for your alarm, down to the millisecond.

• The setInexactRepeating method helps to reduce the battery drain associated with waking

the device on a regular schedule to perform updates. At run time Android will synchronize multiple inexact repeating alarms and trigger them simultaneously.

• Rather than specifying an exact interval, the setInexactRepeating method accepts one of

the following Alarm Manager constants:

1. INTERVAL_FIFTEEN_MINUTES

2. INTERVAL_HALF_HOUR