UNIT V

- Android offers the Service class to create application components that handle long-lived operations and include functionality that doesn’t require a user interface.

- Although Services run without a dedicated GUI, they still execute in the main Thread of the application’s process — just like Activities and Broadcast Receivers.

- To keep your applications responsive, you’ll learn to move time-consuming processes onto background Threads using the Thread and AsyncTask classes.

- INTRODUCING SERVICES
  - Services run invisibly — doing Internet lookups, processing data, updating your Content Providers, fi ring Intents, and triggering Notifications.
  - Services are designed to be longer-lived — specifically, to perform ongoing and potentially time-consuming operations.
  - Services are started, stopped, and controlled from other application components, including Activities, Broadcast Receivers, and other Services.
  - Running Services have a higher priority than inactive or invisible (stopped) Activities, making them less likely to be terminated by the run time’s resource management.

- # Creating and Controlling Services
  - The following sections describe how to create a new Service, and how to start and stop it using Intents with the startService and stopService methods, respectively. Later you'll learn how to bind a Service to an Activity to provide a richer interface.

  - Creating Services
    - To defi ne a Service, create a new class that extends Service. You'll need to override the onCreate and onBind methods,

```
import android.app.Service;
import android.content.Intent;
import android.os.IBinder;

public class MyService extends Service {
  @Override

  public void onCreate() {
    super.onCreate();
    // TODO: Actions to perform when service is created.
  }

  @Override
  public IBinder onBind(Intent intent) {
    // TODO: Replace with service binding implementation.
    return null;
  }
}
```

- After you've constructed a new Service, you must register it in the application manifest.

```
<service android:enabled="true"
        android:name=".MyService"
        android:permission="com.paad.MY_SERVICE_PERMISSION"/>
```

- Executing a Service and Controlling Its Restart Behavior
  - Override the onStartCommand event handler to execute the task (or begin the ongoing operation) encapsulated by your Service. You can also specify your Service's restart behavior within this handler.
  - The onStartCommand method is called whenever the Service is started using startService, so it may be executed several times within a Service's lifetime.
  - Services are launched on the main Application Thread, meaning that any processing done in the onStartCommand handler will happen on the main GUI Thread.
  - The standard pattern for implementing a Service is to create and run a new Thread from onStartCommand to perform the processing in the background, and then stop the Service when it's been completed.

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    startBackgroundTask(intent, startId);
    return Service.START_STICKY;
}
```

- This pattern lets onStartCommand complete quickly, and it enables you to control the restart behavior by returning one of the following Service constants:
  - START_STICKY — Describes the standard behavior, which is similar to the way in which onStart was implemented prior to Android 2.0. If you return this value, onStartCommand will be called any time your Service restarts after being terminated by the run time. Note that on a restart the Intent parameter passed in to onStartCommand will be null.
  - START_NOT_STICKY — This mode is used for Services that are started to process specific actions or commands. Typically, they will use stopSelf to terminate once that command has been completed.
  - START_REDELIVER_INTENT — In some circumstances you will want to ensure that the commands you have requested from your Service are completed
  - This mode is a combination of the fi rst two; if the Service is terminated by the run time, it will restart only if there are pending start calls or the process was killed prior to its calling stopSelf. In the latter case, a call to onStartCommand will be made, passing in the initial Intent whose processing did not properly complete.

- Starting and Stopping Services
  - To start a Service, call startService. Much like Activities, you can either use an action to implicitly start a Service with the appropriate Intent Receiver registered, or you can explicitly specify the Service using its class.
  - If the Service requires permissions that your application does not have, the call to startService will throw a SecurityException
  - In both cases you can pass values in to the Service's onStart handler by adding extras to the Intent which demonstrates both techniques available for starting a Service.

```
private void explicitStart() {
  // Explicitly start My Service
  Intent intent = new Intent(this, MyService.class);
  // TODO Add extras if required.
  startService(intent);
}

private void implicitStart() {
  // Implicitly start a music Service
  Intent intent = new Intent(MyMusicService.PLAY_ALBUM);
  intent.putExtra(MyMusicService.ALBUM_NAME_EXTRA, "United");

  intent.putExtra(MyMusicService.ARTIST_NAME_EXTRA, "Pheonix");
  startService(intent);
}
```

- To stop a Service, use stopService, specifying an Intent that defines the Service to stop

```
// Stop a service explicitly.
stopService(new Intent(this, MyService.class));

// Stop a service implicitly.
Intent intent = new Intent(MyMusicService.PLAY_ALBUM);
stopService(intent);
```

- When your Service has completed the actions or processing for which it was started, you should terminate it by making a call to stopSelf.
- You can call stopSelf either without a parameter to force an immediate stop, or by passing in a startId value to ensure processing has been completed for each instance of startService called so far.

Background processing in android

- Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU.
- •Each part of such program is called a thread.
- •So, threads are light-weight processes with in a process.

- Background processing in Android refers to
- •the “execution of tasks” in “**different threads**” than the Main Thread [or UIThread],
- •where views are inflated and where the user interacts with our app.

Why Background Processing?

- To avoid UI blockages by I/O events and prevent the famous “Application Not Responding” dialog [ANR].
- •*A freezing app means bad UX.*
- •Some operations are not allowed to run in the Main Thread, such as HTTP calls.
- •To improve performance.

- ANR will be triggered for your app when one of the following conditions occur:
- •While your activity is in the foreground, your app has not responded to an input event or Broadcast Receiver within 5 seconds.
- •While you do not have an activity in the foreground, your Broadcast Receiver hasn't finished executing within a considerable amount of time.

Strict mode

- Using StrictMode helps you find accidental I/O operations on the main thread while you’re developing your app.
- •You can use StrictMode at the application or activity level.
- •**Strict Mode is most commonly used to catch accidental disk or network access on the application's main thread**

- **Background Processing Ways**
- •Thread & Runnable [Core Java]
- •AsyncTask [Android]

- Thread implementation in java can be achieved in two ways:
- 1.Extending the java.lang.Threadclass
- 2.Implementing the java.lang.RunnableInterface

```
public class HelloRunnable implements Runnable {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }

}
```

```
public class HelloThread extends Thread {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new HelloThread()).start();
    }

}
```

- **Methods:**
- **onPreExecute()**−Before doing background operation we should show something on screen like a progressbar.
- **doInBackground(Params)**−background operation on background thread. This should not touch on any main thread activities or fragments.
- **onProgressUpdate (Progress…)**− While doing background operation, if you want to update some information on UI, we can use this method.
- **onPostExecute (Result)**− In this method we can update UI of background operation result.

Generic Types in AsyncTask

- **Type Of VarArg Params**− It contains information about what type of params used for execution.
- **2.ProgressValue**−It contains information about progress units.
- **3.ResultValue**− It contains information about result type.

- # USING BACKGROUND THREADS
  - Responsiveness is one of the most critical attributes of a good Android application. To ensure that your application responds quickly to any user interaction or system event, it’s vital that you move all processing and I/O operations off the main application Thread and into a child Thread.
  - It’s important to use background Threads for any nontrivial processing that doesn’t directly interact with the user interface. It’s particularly important to schedule fi le operations, network lookups, database transactions, and complex calculations on a background Thread.
  - Android offers a number of alternatives for moving your processing to the background.
    - You can implement your own Threads and use the Handler class to synchronize with the GUI Thread before updating the UI.
    - Alternatively, the AsyncTask class lets you define an operation to be performed in the background and provides event handlers that enable you to monitor progress and post the results on the GUI Thread.

- # Using AsyncTask to Run Asynchronous Tasks
  - The AsyncTask class implements a best practice pattern for moving your time-consuming operations onto a background Thread and synchronizing with the UI Thread for updates and when the processing is complete.
  - It offers the convenience of event handlers synchronized with the GUI Thread to let you update Views and other UI elements to report progress or publish results when your task is complete.
  - AsyncTask handles all the Thread creation, management, and synchronization, enabling you to create an asynchronous task consisting of processing to be done in the background and UI updates to be performed both during the processing, and once it’s complete

- Creating New Asynchronous Tasks
  - Each AsyncTask implementation can specify parameter types that will be used for input parameters, the progress-reporting values, and result values. If you don't need or want to take input parameters, update progress, or report a fi nal result, simply specify Void for any or all the types required.

```
private class MyAsyncTask extends AsyncTask<String, Integer, String> {
  @Override
  protected String doInBackground(String... parameter) {
    // Moved to a background thread.
    String result = "";
    int myProgress = 0;

    int inputLength = parameter[0].length();

    // Perform background processing task, update myProgress]
    for (int i = 1; i <= inputLength; i++) {
      myProgress = i;
      result = result + parameter[0].charAt(inputLength-i);
      try {
        Thread.sleep(100);
      } catch (InterruptedException e) { }
      publishProgress(myProgress);
    }

    // Return the value to be passed to onPostExecute
    return result;
  }

  @Override
  protected void onProgressUpdate(Integer... progress) {
    // Synchronized to UI thread.
    // Update progress bar, Notification, or other UI elements
    asyncProgress.setProgress(progress[0]);
  }
```

```
  @Override
  protected void onPostExecute(String result) {
    // Synchronized to UI thread.
    // Report results via UI update, Dialog, or notifications
    asyncTextView.setText(result);
  }
}
```

- Your subclass should also override the following event handlers:

  - ‰ doInBackground — This method will be executed on the background Thread, so place your long-running code here, and don't attempt to interact with UI objects from within this handler. It takes a set of parameters of the type defined in your class implementation.

    You can use the publishProgress method from within this handler to pass parameter values to the onProgressUpdate handler, and when your background task is complete, you can return the final result as a parameter to the onPostExecute handler, which can update the UI accordingly.

  - ‰ onProgressUpdate — Override this handler to update the UI with interim progress updates. This handler receives the set of parameters passed in to publishProgress (typically from within the doInBackground handler). This handler is synchronized with the GUI Thread when executed, so you can safely modify UI elements.

  - ‰ onPostExecute — When doInBackground has completed, the return value from that method is passed in to this event handler. Use this handler to update the UI when your asynchronous task has completed. This handler is synchronized with the GUI Thread when executed, so you can safely modify UI elements.

- Running Asynchronous Tasks
  - After you've implemented an asynchronous task, execute it by creating a new instance and calling execute, You can pass in a number of parameters, each of the type specifi ed in your implementation.

    - String input = "redrum ... redrum";
    - new MyAsyncTask().execute(input);

- USING ALARMS
  - Alarms are a means of firing Intents at predetermined times or intervals.
  - Unlike Timers, Alarms operate outside the scope of your application, so you can use them to trigger application events or actions even after your application has been closed.
  - Alarms are particularly powerful when used in combination with Broadcast Receivers, enabling you to set Alarms that fire broadcast Intents, start Services, or even open Activities, without your application needing to be open or running.

  - 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 follows:

    - AlarmManager alarmManager = (AlarmManager)getSystemService(Context.ALARM_SERVICE);

- Creating, Setting, and Canceling Alarms
  - To create a new one-shot Alarm, use the set method and specify an alarm type, a trigger time, and a Pending Intent to fi re when the Alarm triggers. If the trigger time you specify for the Alarm occurs in the past, the Alarm will be triggered immediately.

  - The following four alarm types are available:

    - ‰ RTC_WAKEUP — Wakes the device from sleep to fire the Pending Intent at the clock time specified.
    - ‰ RTC — Fires the Pending Intent at the time specified but does not wake the device.
    - ‰ 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.
    - ‰ ELAPSED_REALTIME_WAKEUP — Wakes the device from sleep and fires the Pending Intent after a specified length of time has passed since device boot.

```
// Get a reference to the Alarm Manager
AlarmManager alarmManager =
 (AlarmManager)getSystemService(Context.ALARM_SERVICE);

// Set the alarm to wake the device if sleeping.
int alarmType = AlarmManager.ELAPSED_REALTIME_WAKEUP;

// Trigger the device in 10 seconds.
long timeOrLengthofWait = 10000;

// Create a Pending Intent that will broadcast and action
String ALARM_ACTION = "ALARM_ACTION";
Intent intentToFire = new Intent(ALARM_ACTION);
PendingIntent alarmIntent = PendingIntent.getBroadcast(this, 0,
  intentToFire, 0);

// Set the alarm
alarmManager.set(alarmType, timeOrLengthofWait, alarmIntent);
```

- When the Alarm goes off, the Pending Intent you specified will be broadcast. Setting a second Alarm using the same Pending Intent replaces the preexisting Alarm.
- To cancel an Alarm, call cancel on the Alarm Manager, passing in the Pending Intent you no longer want to trigger.

Setting Repeating Alarms

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

- To set a repeating alarm, use the setRepeating or setInexactRepeating method on the Alarm Manager. Both methods support an alarm type, an initial trigger time, and a Pending Intent to fi re when the alarm triggers.

- Use setRepeating when you need fi ne-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:
  - INTERVAL_FIFTEEN_MINUTES
  - INTERVAL_HALF_HOUR
  - INTERVAL_HOUR
  - INTERVAL_HALF_DAY
  - INTERVAL_DAY

```
// Get a reference to the Alarm Manager
AlarmManager alarmManager =
 (AlarmManager)getSystemService(Context.ALARM_SERVICE);

// Set the alarm to wake the device if sleeping.
int alarmType = AlarmManager.ELAPSED_REALTIME_WAKEUP;

// Schedule the alarm to repeat every half hour.
long timeOrLengthofWait = AlarmManager.INTERVAL_HALF_HOUR;

// Create a Pending Intent that will broadcast and action
String ALARM_ACTION = "ALARM_ACTION";
Intent intentToFire = new Intent(ALARM_ACTION);
PendingIntent alarmIntent = PendingIntent.getBroadcast(this, 0,
  intentToFire, 0);

// Wake up the device to fire an alarm in half an hour, and every
// half-hour after that.
alarmManager.setInexactRepeating(alarmType,
                                 timeOrLengthofWait,
                                 timeOrLengthofWait,
                                 alarmIntent);
```

• INTRODUCING NOTIFICATIONS

- Notifications are handled by the Notification Manager and currently have the ability to
  - Display a status bar icon
  - Flash the lights/LEDs
  - Vibrate the phone
  - Sound audible alerts (ringtones, Media Store audio)
  - Display additional information within the notification tray
  - Broadcast Intents using interactive controls from within the notification tray
- Notifications are the preferred mechanism for invisible application components (Broadcast Receivers, Services, and inactive Activities) to alert users that events have occurred that may require attention.
- In some environment it’s important that your applications be able to alert users when specific events occur that require their attention.

Notification Overview

- A notification is a message that Android displays outside your app's UI to provide the user with reminders, communication from other people, or other timely information from your app.

- Users can tap the notification to open your app or take an action directly from the notification.

Appearances on a device

- Notifications appear to users in different locations and formats, such as an icon in the status bar, a more detailed entry in the notification drawer, as a badge on the app's icon, and on paired wearables automatically.

Status bar and notification drawer

When you issue a notification, it first appears as an icon in the status bar.

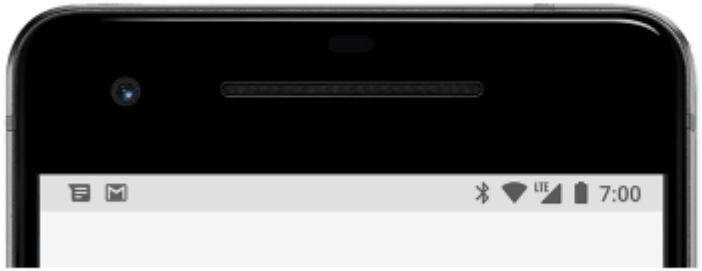


Figure 1. Notification icons appear on the left side of the status bar

Users can swipe down on the status bar to open the notification drawer, where they can view more details and take actions with the notification.

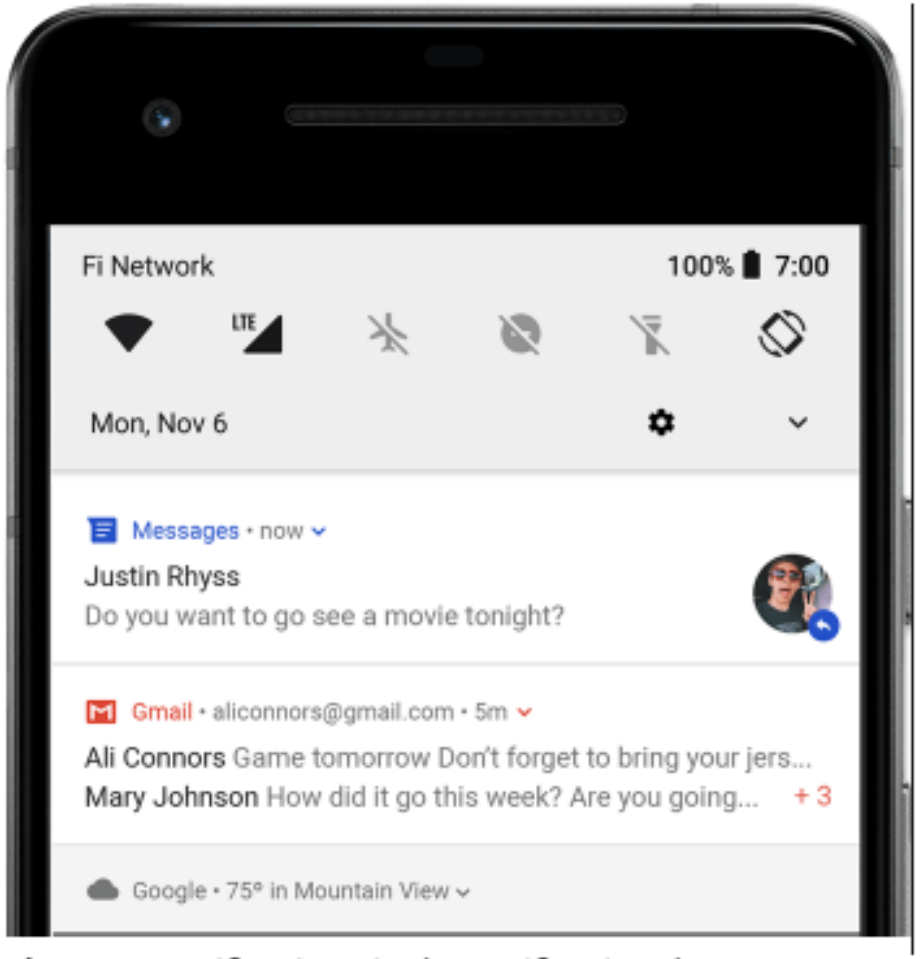


Figure 2. Notifications in the notification drawer

- Users can drag down on a notification in the drawer to reveal the expanded view, which shows additional content and action buttons, if provided. Starting in Android 13, this expanded view includes a button that allows users to stop an app that has ongoing foreground services.

Heads-up notification

Beginning with Android 5.0, notifications can briefly appear in a floating window called a *heads-up notification*. This behavior is normally for important notifications that the user should know about immediately, and it appears only if the device is unlocked.

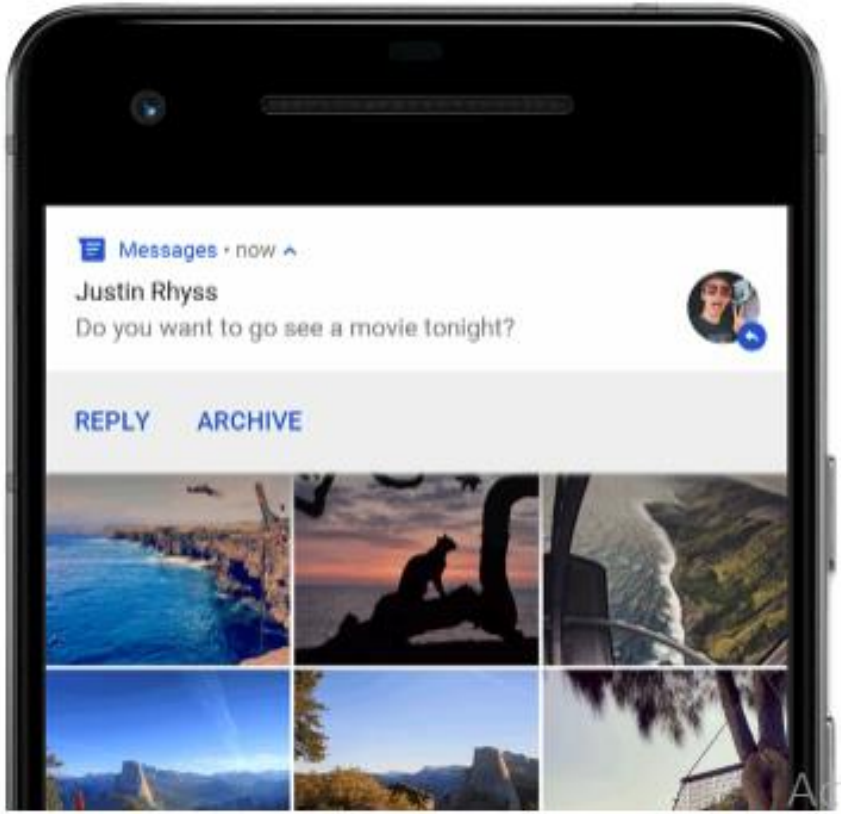


Figure 3. A heads-up notification appears in front of the foreground app

- The heads-up notification appears the moment your app issues the notification and it disappears after a moment, but remains visible in the notification drawer as usual.

- Example conditions that might trigger heads-up notifications include the following:

- The user's activity is in fullscreen mode (the app uses fullScreenIntent).
- The notification has high priority and uses ringtones or vibrations on devices running Android 7.1 (API level 25) and lower.
- The notification channel has high importance on devices running Android 8.0 (API level 26) and higher.

Notification anatomy

- The design of a notification is determined by system templates—your app simply defines the contents for each portion of the template. Some details of the notification appear only in the expanded view.

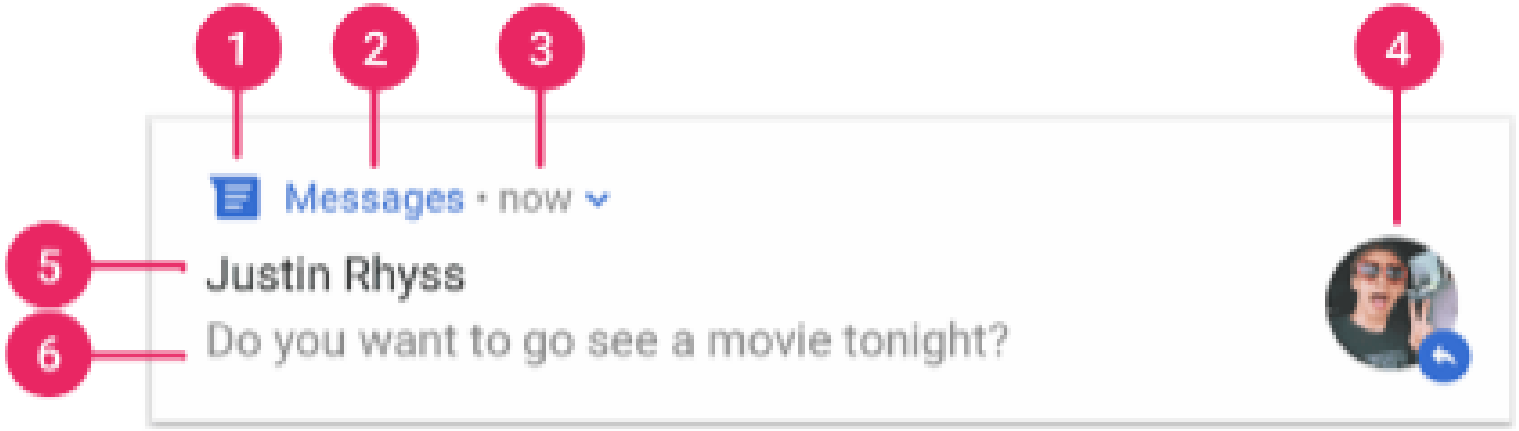


Figure 7. A notification with basic details

The most common parts of a notification are indicated in figure 7 as follows:

1. Small icon: This is required and set with `setSmallIcon()`.
2. App name: This is provided by the system.
3. Time stamp: This is provided by the system but you can override with `setWhen()` or hide it with `setShowWhen(false)`.
4. Large icon: This is optional (usually used only for contact photos; do not use it for your app icon) and set with `setLargeIcon()`.
5. Title: This is optional and set with `setContentTitle()`.
6. Text: This is optional and set with `setContentText()`.

Notification channels

- Starting in Android 8.0 (API level 26), all notifications must be assigned to a channel or it will not appear.
- By categorizing notifications into channels, users can disable specific notification channels for your app (instead of disabling *all* your notifications), and users can control the visual and auditory options for each channel—all from the Android system settings (figure 11).
- Users can also long-press a notification to change behaviors for the associated channel.

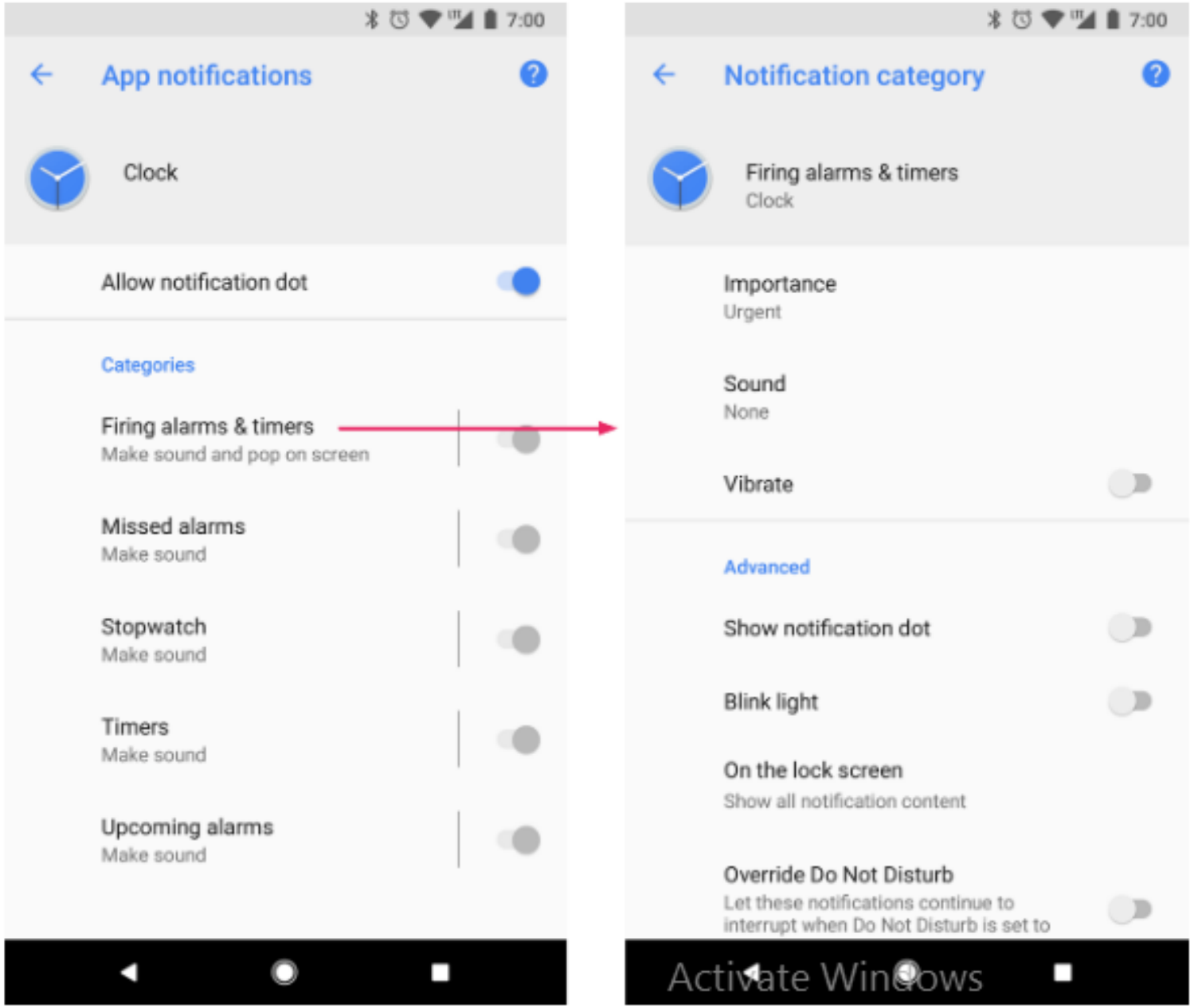


Figure 11. Notification settings for Clock app and one of its channels

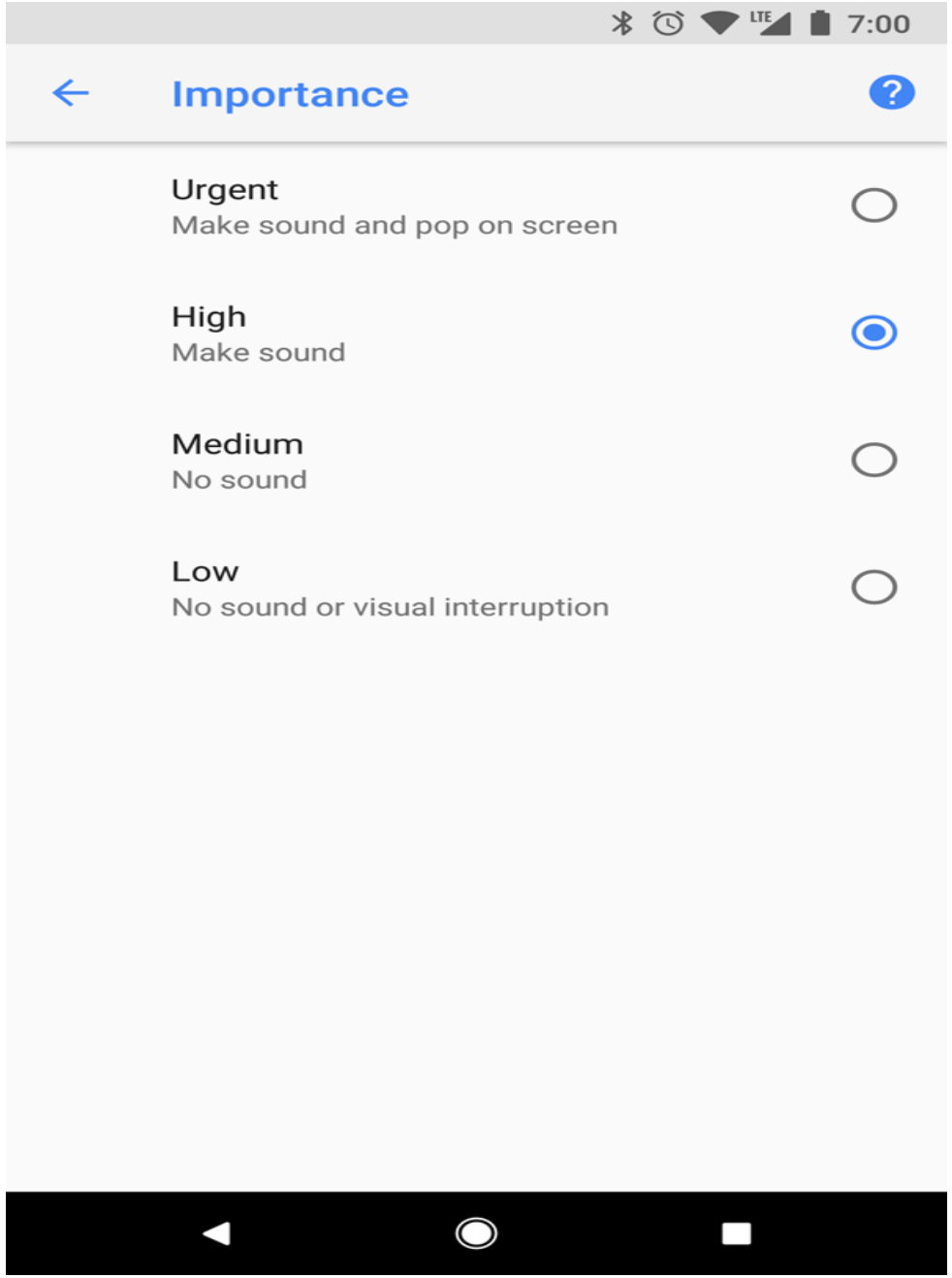
- On devices running Android 7.1 (API level 25) and lower, users can manage notifications on a per-app basis only (effectively each app only has one channel on Android 7.1 and lower).

- One app can have multiple notification channels—a separate channel for each type of notification the app issues.
- An app can also create notification channels in response to choices made by users of your app.
- For example, you may set up separate notification channels for each conversation group created by a user in a messaging app.
- The channel is also where you specify the importance level for your notifications on Android 8.0 and higher.
- So all notifications posted to the same notification channel have the same behavior.

Notification importance

- Android uses the importance of a notification to determine how much the notification should interrupt the user (visually and audibly).
- The higher the importance of a notification, the more interruptive the notification will be.

- On Android 8.0 (API level 26) and above, importance of a notification is determined by the importance of the channel the notification was posted to.
- Users can change the importance of a notification channel in the system settings (figure 12).
- On Android 7.1 (API level 25) and below, importance of each notification is determined by the notification's priority.



- **Figure 12.** Users can change the importance of each channel on Android 8.0 and higher

- The possible importance levels are the following:
- Urgent: Makes a sound and appears as a heads-up notification.
- High: Makes a sound.
- Medium: No sound.
- Low: No sound and does not appear in the status bar.
- All notifications, regardless of importance, appear in non-interruptive system UI locations, such as in the notification drawer and as a badge on the launcher icon (though you can modify the appearance of the notification badge).

Notification compatibility

- Since Android 1.0, the notification system UI and the notification-related APIs have continually evolved.
- To use the latest notification API features while still supporting older devices, use the support library notification API: NotificationCompat and its subclasses, as well as NotificationManagerCompat.
- This will allow you to avoid writing conditional code to check API levels because these APIs handle that for you.

- setAutoCancel(Boolean autoCancel)
- Make this notification automatically dismissed when the user touches it.
- setContentIntent (PendingIntent intent)
- Supply a PendingIntent to be sent when the notification is clicked.
- setContentText (CharSequence text)
- Set the second line of text in the platform notification template.
- setContentTitle (CharSequence title)
- Set the first line of text in the platform notification template.
- setLargeIcon (Bitmap b/Icon i)
- Add a large icon to the notification content view.
- setSmallIcon (int icon)
- Set the small icon resource, which will be used to represent the notification in the status bar.

- Notifications can be persisted through insistent repetition, being marked ongoing, or simply by displaying an icon on the status bar. Status bar icons can be updated regularly or expanded to show additional information using the expanded notification tray

- Introducing the Notification Manager
  - The NotificationManager is a system Service used to manage Notifications. Get a reference to it using the getSystemService method

```
String svcName = Context.NOTIFICATION_SERVICE;

NotificationManager notificationManager;
notificationManager = (NotificationManager)getSystemService(svcName);
```

  - Using the Notification Manager, you can trigger new Notifications, modify existing ones, or cancel those that are no longer required.

- Creating Notifications
  - Android offers a number of ways to convey information to users using Notifications:
    - Status bar icon
    - Sounds, lights, and vibrations
    - Details displayed within the extended notification tray
  - **Creating a Notification and Configuring the Status Bar Display**
    - Start by creating a new Notification object, passing in an icon to display on the status bar along with the ticker text to display on the status bar when the Notification is triggered.

```
// Choose a drawable to display as the status bar icon
int icon = R.drawable.icon;
// Text to display in the status bar when the notification is launched
String tickerText = "Notification";
// The extended status bar orders notification in time order
long when = System.currentTimeMillis();

Notification notification = new Notification(icon, tickerText, when);
```

- The ticker text should be a short summary that describes what you are notifying the user of (for example, an SMS message or email subject line).

- You also need to specify the timestamp of the Notification; the Notification Manager will sort Notifications in this order.

- You can also set the Notification object's number property to display the number of events a status bar icon represents.

- Android 3.0 (API level 11) introduced the Notification.Builder class to simplify the process of adding device ring, flash, and vibrate.

- Using the Default Notification Sounds, Lights, and Vibrations
  - The simplest and most consistent way to add sounds, lights, and vibrations to your Notifications is to use the default settings. Using the defaults property, you can combine the following constants:
    - Notification.DEFAULT_LIGHTS
    - Notification.DEFAULT_SOUND
    - Notification.DEFAULT_VIBRATE

```
notification.defaults = Notification.DEFAULT_SOUND | Notification.DEFAULT_VIBRATE;
```

  - If you want to use all the default values, you can use the Notification.DEFAULT_ALL constant.

- Making Sounds
  - Most native phone events, from incoming calls to new messages and low battery, are announced by audible ringtones.

```
Uri ringURI = RingtoneManager.getDefaultUri(RingtoneManager.TYPE_NOTIFICATION);
notification.sound = ringURI;
```

- Vibrating the Device
  - You can use the device's vibrator to execute a vibration pattern specific to your Notification. Android lets you control the pattern of a vibration
    - <uses-permission android:name="android.permission.VIBRATE"/>
  - To set a vibration pattern, assign a long[] to the Notifi cation's vibrate property.
    - long[] vibrate = new long[] { 1000, 1000, 1000, 1000, 1000 };
    - notification.vibrate = vibrate;

CONTROLLING DEVICE VIBRATION

- In some circumstances, you may want to vibrate the device independently of Notifications
- For example, vibrating the device is an excellent way to provide haptic user feedback and is particularly popular as a feedback mechanism for games.
- To control device vibration, your applications needs the VIBRATE permission:

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

- Device vibration is controlled through the Vibrator Service, accessible via the getSystemService method:

```
String vibratorService = Context.VIBRATOR_SERVICE;
Vibrator vibrator = (Vibrator)getSystemService(vibratorService);
```

- Call vibrate to start device vibration; you can pass in either a vibration duration or a pattern of alternating vibration/pause sequences along with an optional index parameter that repeats the pattern starting at the index specified:

```
long[] pattern = {1000, 2000, 4000, 8000, 16000 };
vibrator.vibrate(pattern, 0); // Execute vibration pattern.
vibrator.vibrate(1000); // Vibrate for 1 second.
```

- To cancel vibration, call cancel; exiting your application automatically cancels any vibration it has initiated.

```
vibrator.cancel();
```

- 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, whereas the ledOffMS and ledOnMS properties let you set the frequency and pattern of the flashing LED. You can turn on the LED by setting the ledOnMS property to 1 and the ledOffMS property to 0, or turn it off by setting both properties to 0.
  - After configuring the LED settings, you must also add the FLAG_SHOW_LIGHTS flag to the Notification's flags property.

```
notification.ledARGB = Color.RED;
notification.ledOffMS = 0;
notification.ledOnMS = 1;
notification.flags = notification.flags | Notification.FLAG_SHOW_LIGHTS;
```

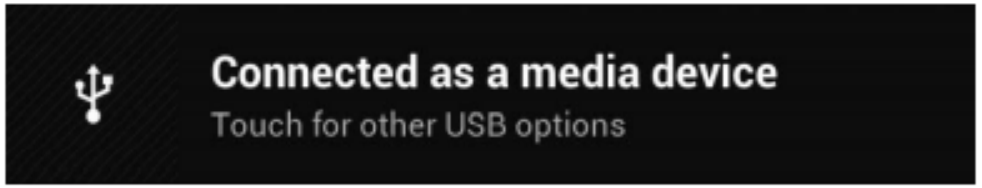
- Using the Notification Builder
  - The Notification Builder, introduced in Android 3.0 (API level 11) to simplify the process of configuring the flags, options, content, and layout of Notifications, is the preferred alternative when constructing Notifications for newer Android platforms.

```
Notification.Builder builder =
  new Notification.Builder(MyActivity.this);

builder.setSmallIcon(R.drawable.ic_launcher)
       .setTicker("Notification")
       .setWhen(System.currentTimeMillis())
       .setDefaults(Notification.DEFAULT_SOUND |
                    Notification.DEFAULT_VIBRATE)
       .setSound(
          RingtoneManager.getDefaultUri(
            RingtoneManager.TYPE_NOTIFICATION))
       .setVibrate(new long[] { 1000, 1000, 1000, 1000, 1000 })
       .setLights(Color.RED, 0, 1);

Notification notification = builder.getNotification();
```

- Setting and Customizing the Notification Tray UI
  - You can configure the appearance of the Notification within the extended notification tray in a number of ways:
    - ‰ Use the setLatestEventInfo method to update the details displayed in the standard notification tray display.
    - ‰ Use the Notification Builder to create and control one of several alternative notification tray UIs.
    - ‰ Set the contentView and contentIntent properties to assign a custom UI for the extended status display using a Remote Views object.
    - ‰ From Android 3.0 (API level 11) onward, you can assign Broadcast Intents to each View within the Remote Views object that describes your custom UI to make them fully interactive.

- Using the Standard Notification UI
  - The simplest approach is to use the setLatestEventInfo method to specify the title and text fields used to populate the default notification tray layout.

```
notification.setLatestEventInfo(context,
                                expandedTitle,
                                expandedText,
                                launchIntent);
```



  - Android 3.0 (API level 11) expanded the size used for each Notification, introducing support for a larger icon to be displayed within the notification tray. You can assign the large icon by assigning it to the largeIcon property of your Notification.

  - Alternatively, you can use the Notification Builder to populate these details

```
builder.setSmallIcon(R.drawable.ic_launcher)
       .setTicker("Notification")
       .setWhen(System.currentTimeMillis())
       .setContentTitle("Title")
       .setContentText("Subtitle")
       .setContentInfo("Info")
       .setLargeIcon(myIconBitmap)
       .setContentIntent(pendingIntent);
```

- The Notification Builder also provides support for displaying a progress bar within your Notification. Using the setProgress method.


```
builder.setSmallIcon(R.drawable.ic_launcher)
       .setTicker("Notification")
       .setWhen(System.currentTimeMillis())
       .setContentTitle("Progress")
       .setProgress(100, 50, false)
       .setContentIntent(pendingIntent);
```

INTRODUCING ANDROID TEXT-TO-SPEECH

- The text-to-speech (TTS) libraries, also known as speech synthesis, enable you to output synthesized speech from within your applications, allowing them to “talk” to your users.

- Before using the TTS engine, it’s good practice to confirm the language packs are installed.

- To check for the TTS libraries, start a new Activity for a result using the ACTION_CHECK_TTS_DATA action from the TextToSpeech.Engine class:

```
Intent intent = new Intent(TextToSpeech.Engine.ACTION_CHECK_TTS_DATA);
startActivityForResult(intent, TTS_DATA_CHECK);
```

- The onActivityResult handler receives CHECK_VOICE_DATA_PASS if the voice data has been installed successfully.
- If the voice data is not currently available, start a new Activity using the ACTION_INSTALL_TTS_DATA action from the TTS Engine class to initiate its installation.

```
Intent installVoice = new Intent(Engine.ACTION_INSTALL_TTS_DATA);
startActivity(installVoice);
```

- After confirming the voice data is available, you need to create and initialize a new TextToSpeech instance.
- Note that you cannot use the new Text To Speech object until initialization is complete.
- Pass an OnInitListener into the constructor that will be fired when the TTS engine has been initialized.

```
boolean ttsIsInit = false;
TextToSpeech tts = null;

protected void onActivityResult(int requestCode,
                                int resultCode, Intent data) {
  if (requestCode == TTS_DATA_CHECK) {
    if (resultCode == Engine.CHECK_VOICE_DATA_PASS) {
      tts = new TextToSpeech(this, new OnInitListener() {
       public void onInit(int status) {
         if (status == TextToSpeech.SUCCESS) {
           ttsIsInit = true;
           // TODO Speak!
         }
       }
      });
    }
  }
}
```

- After initializing Text To Speech, you can use the speak method to synthesize voice data using the default device audio output:

```
HashMap parameters = null;
tts.speak("Hello, Android", TextToSpeech.QUEUE_ADD, parameters);
```

- The speak method enables you to specify a parameter either to add the new voice output to the existing queue or to flush the queue and start speaking immediately.

- You can affect the way the voice output sounds using the setPitch and setSpeechRate methods.

- You can also change the pronunciation of your voice output using the setLanguage method.
- This method takes a Locale parameter to specify the country and language of the text to speak. This affects the way the text is spoken to ensure the correct language and pronunciation models are used.

- When you have finished speaking, use stop to halt voice output and shutdown to free the TTS resources:

```
tts.stop();
tts.shutdown();
```

- Whether the TTS voice library is installed, initializes a new TTS engine, and uses it to speak in UK English.

```
private static int TTS_DATA_CHECK = 1;

private TextToSpeech tts = null;
private boolean ttsIsInit = false;

private void initTextToSpeech() {
  Intent intent = new Intent(Engine.ACTION_CHECK_TTS_DATA);
  startActivityForResult(intent, TTS_DATA_CHECK);
}

protected void onActivityResult(int requestCode,
                                int resultCode, Intent data) {
  if (requestCode == TTS_DATA_CHECK) {
    if (resultCode == Engine.CHECK_VOICE_DATA_PASS) {
      tts = new TextToSpeech(this, new OnInitListener() {
       public void onInit(int status) {
         if (status == TextToSpeech.SUCCESS) {
           ttsIsInit = true;
           if (tts.isLanguageAvailable(Locale.UK) >= 0)
```

```
            tts.setLanguage(Locale.UK);
            tts.setPitch(0.8f);
            tts.setSpeechRate(1.1f);
            speak();
          }
        }
      });
    } else {
      Intent installVoice = new Intent(Engine.ACTION_INSTALL_TTS_DATA);
      startActivity(installVoice);
    }
  }
}

private void speak() {
  if (tts != null && ttsIsInit) {
    tts.speak("Hello, Android", TextToSpeech.QUEUE_ADD, null);
  }
}

@Override
public void onDestroy() {
  if (tts != null) {
    tts.stop();
    tts.shutdown();
  }
  super.onDestroy();
}
```

- # USING SPEECH RECOGNITION
  - Android supports voice input and speech recognition using the RecognizerIntent class. This API enables you to accept voice input into your application using the standard voice input dialog.
  - To initialize voice recognition, call startNewActivityForResult, passing in an Intent that specifies the RecognizerIntent.ACTION_RECOGNIZE_SPEECH or RecognizerIntent.ACTION_WEB_SEARCH actions.
  - The launch Intent must include the RecognizerIntent.EXTRA_LANGUAGE_MODEL extra to specify the language model used to parse the input audio.
  - This can be either LANGUAGE_MODEL_FREE_FORM or LANGUAGE_MODEL_WEB_SEARCH; both are available as static constants from the RecognizerIntent class.

- You can also specify a number of optional extras to control the language, potential result count, and display prompt using the following Recognizer Intent constants:
  - ‰ EXTRA_LANGUAGE — Specifies a language constant from the Locale class to use an input language other than the device default. You can find the current default by calling the static getDefault method on the Locale class.
  - ‰ EXTRA_MAXRESULTS — Uses an integer value to limit the number of potential recognition results returned.
  - ‰ EXTRA_PROMPT — Specifies a string that displays in the voice input dialog (shown in Figure 11-6) to prompt the user to speak.

- Using Speech Recognition for Voice Input
  - When using voice recognition to receive the spoken words, call startNewActivityForResult using the RecognizerIntent.ACTION_RECOGNIZE_SPEECH action

```
Intent intent = new Intent(RecognizerIntent.ACTION_RECOGNIZE_SPEECH);
// Specify free form input
intent.putExtra(RecognizerIntent.EXTRA_LANGUAGE_MODEL,
                RecognizerIntent.LANGUAGE_MODEL_FREE_FORM);
intent.putExtra(RecognizerIntent.EXTRA_PROMPT,
                "or forever hold your peace");
intent.putExtra(RecognizerIntent.EXTRA_MAX_RESULTS, 1);
intent.putExtra(RecognizerIntent.EXTRA_LANGUAGE, Locale.ENGLISH);
startActivityForResult(intent, VOICE_RECOGNITION);
```

  - When the user finishes speaking, the speech recognition engine analyzes and processes the resulting audio and then returns the results through the onActivityResult handler as an Array List of strings in the EXTRA_RESULTS extra.

```
@Override
protected void onActivityResult(int requestCode,
                                int resultCode,
                                Intent data) {

    if (requestCode == VOICE_RECOGNITION && resultCode == RESULT_OK) {
      ArrayList<String> results;

      results =
        data.getStringArrayListExtra(RecognizerIntent.EXTRA_RESULTS);

      float[] confidence;

      String confidenceExtra = RecognizerIntent.EXTRA_CONFIDENCE_SCORES;
      confidence =
        data.getFloatArrayExtra(confidenceExtra);

      // TODO Do something with the recognized voice strings
    }
    super.onActivityResult(requestCode, resultCode, data);
  }
```

- PLAYING AUDIO AND VIDEO
  - Android 4.0.3 (API level 15) supports the following multimedia formats for playback as part of the base framework. Note that some devices may support playback of additional fi le formats:

- **Audio**
  - AAC LC/LTP
  - HE-AACv1 (AAC+)
  - HE-AACv2 (Enhanced AAC+)
  - AMR-NB
  - AMR-WB
  - MP3
  - MIDI
  - Ogg Vorbis
  - PCM/WAVE
  - FLAC (on devices running Android 3.1 or above)
- **Image**
  - JPEG
  - PNG
  - WEBP (on devices running Android 4.0 or above)
  - GIF
  - BMP

- **Video**
  - H.263
  - H.264 AVC
  - MPEG-4 SP
  - VP8 (on devices running Android 2.3.3 or above)

The following network protocols are supported for streaming media:

- RTSP (RTP, SDP)
- HTTP/HTTPS progressive streaming
- HTTP/HTTPS live streaming (on devices running Android 3.0 or above)

- Introducing the Media Player
  - The playback of audio and video within Android applications is handled primarily through the MediaPlayer class.
  - Using the Media Player, you can play media stored in application resources, local files, Content Providers, or streamed from a network URL.

  - The Media Player's management of audio and video fi les and streams is handled as a state machine.
    - 1. Initialize the Media Player with media to play.
    - 2. Prepare the Media Player for playback.
    - 3. Start the playback.
    - 4. Pause or stop the playback prior to its completing.
    - 5. The playback is complete.

- Preparing Audio for Playback
  - Initializing Audio Content for Playback
    - To play back audio content using a Media Player, you need to create a new Media Player object and set the data source of the audio in question. You can do this by using the static create method, passing in the Activity Context and any one of the following audio sources:

      - A resource identifier (typically for an audio file stored in the `res/raw` resource folder)
      - A URI to a local file (using the `file://` schema)
      - A URI to an online audio resource (as a URL)
      - A URI to a row within a Content Provider that returns an audio file

```
// Load an audio resource from a package resource.
MediaPlayer resourcePlayer =
  MediaPlayer.create(this, R.raw.my_audio);

// Load an audio resource from a local file.
MediaPlayer filePlayer = MediaPlayer.create(this,
  Uri.parse("file:///sdcard/localfile.mp3"));

// Load an audio resource from an online resource.
MediaPlayer urlPlayer = MediaPlayer.create(this,
  Uri.parse("http://site.com/audio/audio.mp3"));

// Load an audio resource from a Content Provider.
MediaPlayer contentPlayer = MediaPlayer.create(this,
  Settings.System.DEFAULT_RINGTONE_URI);
```

- Alternatively, you can use the setDataSource method on an existing Media Player instance

- This method accepts a fi le path, Content Provider URI, streaming media URL path, or File Descriptor. When using the setDataSource method, it is vital that you call prepare on the Media Player before you begin playback.

```
MediaPlayer mediaPlayer = new MediaPlayer();
mediaPlayer.setDataSource("/sdcard/mydopetunes.mp3");
mediaPlayer.prepare();
```

- Preparing Video for Playback
  - There are two alternatives for the playback of video content. The first technique, using the VideoView class, encapsulates the creation of a Surface and allocation and preparation of video content using a Media Player.
  - The second technique allows you to specify your own Surface and manipulate the underlying Media Player instance directly.

- Playing Video Using the Video View
  - The simplest way to play back video is to use the Video View. The Video View includes a Surface on which the video is displayed and encapsulates and manages a Media Player instance that handles the playback.

  - After placing the Video View within the UI, get a reference to it within your code. You can then assign a video to play by calling its setVideoPath or setVideoURI methods to specify the path to a local fi le, or the URI of either a Content Provider or remote video stream:

```
final VideoView videoView = (VideoView)findViewById(R.id.videoView);

// Assign a local file to play
videoView.setVideoPath("/sdcard/mycatvideo.3gp");

// Assign a URL of a remote video stream
videoView.setVideoUri(myAwesomeStreamingSource);
```

- When the video is initialized, you can control its playback using the start, stopPlayback, pause, and seekTo methods.

- The Video View also includes the setKeepScreenOn method to apply a screen Wake Lock that will prevent the screen from being dimmed while playback is in progress without requiring a special permission.

```
// Get a reference to the Video View.
final VideoView videoView = (VideoView)findViewById(R.id.videoView);

// Configure the video view and assign a source video.
videoView.setKeepScreenOn(true);
videoView.setVideoPath("/sdcard/mycatvideo.3gp");

// Attach a Media Controller
MediaController mediaController = new MediaController(this);
videoView.setMediaController(mediaController);
```

- Controlling Media Player Playback
    - When a Media Player is prepared, call start to begin playback of the associated media:
        - mediaPlayer.start();
    - Use the stop and pause methods to stop or pause playback, respectively.

- USING THE CAMERA FOR TAKING PICTURES
  - Using Intents to Take Pictures
    - The easiest way to take a picture from within your application is to fi re an Intent using the MediaStore.ACTION_IMAGE_CAPTURE action:
      - startActivityForResult(new Intent(MediaStore.ACTION_IMAGE_CAPTURE), TAKE_PICTURE);
    - This launches a Camera application to take the photo, providing your users with the full suite of camera functionality without you having to rewrite the native Camera application.
    - Once users are satisfi ed with the image, the result is returned to your application within the Intent received by the onActivityResult handler.
    - By default, the picture taken will be returned as a thumbnail, available as a raw bitmap within the data extra within the returned Intent.

- To obtain a full image, you must specify a target fi le in which to store it, encoded as a URI passed in using the MediaStore.EXTRA_OUTPUT extra in the launch Intent

```
// Create an output file.
File file = new File(Environment.getExternalStorageDirectory(),
                     "test.jpg");
Uri outputFileUri = Uri.fromFile(file);

// Generate the Intent.
Intent intent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
intent.putExtra(MediaStore.EXTRA_OUTPUT, outputFileUri);

// Launch the camera app.
startActivityForResult(intent, TAKE_PICTURE);
```

- The full-size image taken by the camera will then be saved to the specifi ed location.

- RECORDING VIDEO
  - Using Intents to Record Video
    - The easiest, and best practice, way to initiate video recording is using the MediaStore.ACTION_VIDEO_CAPTURE action Intent.
    - Starting a new Activity with this Intent launches the native video recorder, allowing users to start, stop, review, and retake their video. When they're satisfied, a URI to the recorded video is provided to your Activity as the data parameter of the returned Intent:
  - The video capture action Intent can contain the following three optional extras:
    - MediaStore.EXTRA_OUTPUT — By default, the video recorded by the video capture action will be stored in the default Media Store. If you want to record it elsewhere, you can specify an alternative URI using this extra.
    - MediaStore.EXTRA_VIDEO_QUALITY — The video capture action allows you to specify an image quality using an integer value. There are currently two possible values: 0 for low (MMS) quality videos, or 1 for high (full resolution) videos. By default, the high-resolution mode is used.
    - MediaStore.EXTRA_DURATION_LIMIT — The maximum length of the recorded video (in seconds).

```
private static final int RECORD_VIDEO = 0;

private void startRecording() {
  // Generate the Intent.
  Intent intent = new Intent(MediaStore.ACTION_VIDEO_CAPTURE);

  // Launch the camera app.
  startActivityForResult(intent, RECORD_VIDEO);
}

@Override
protected void onActivityResult(int requestCode,
                                int resultCode, Intent data) {
  if (requestCode == RECORD_VIDEO) {
    VideoView videoView = (VideoView)findViewById(R.id.videoView);
    videoView.setVideoURI(data.getData());
    videoView.start()
  }
}
```

- Using the Media Recorder to Record Video
  - You can use the MediaRecorder class to record audio and/or video fi les that can be used in your own applications or added to the Media Store.

  - To record any media in Android, your application needs the CAMERA and RECORD_AUDIO and/or RECORD_VIDEO permissions as applicable:

```
<uses-permission android:name=”android.permission.RECORD_AUDIO”/>
<uses-permission android:name=”android.permission.RECORD_VIDEO”/>
<uses-permission android:name=”android.permission.CAMERA”/>
```

  - The Media Recorder lets you specify the audio and video source, the output fi le format, and the audio and video encoders to use when recording your file. Android 2.2 (API level 8) introduced the concept of profiles, which can be used to apply a predefined set of Media Recorder configurations.

  - Much like the Media Player, the Media Recorder manages recording as a state machine. This means that the order in which you confi gure and manage the Media Recorder is important.

- In the simplest terms, the transitions through the state machine can be described as follows:
    - 1. Create a new Media Recorder.
    - 2. Unlock the Camera and assign it to the Media Recorder.
    - 3. Specify the input sources to record from.
    - 4. Select a profile to use for Android 2.2 and above, or define the output format and specify the audio and video encoder, frame rate, and output size.
    - 5. Select an output fi le.
    - 6. Assign a preview Surface.
    - 7. Prepare the Media Recorder for recording.
    - 8. Record.
    - 9. End the recording.

- When you finish recording your media, call release on your Media Recorder object to free the associated resources:

```
mediaRecorder.release();
```

- Telephony and SMS

- **USING SENSORS AND THE SENSOR MANAGER**
- The Sensor Manager is used to manage the sensor hardware available on Android devices. Use getSystemService to return a reference to the Sensor Manager Service.

- String service_name = Context.SENSOR_SERVICE;
- SensorManager sensorManager = (SensorManager)getSystemService(service_name);

- The Sensor class includes a set of constants that describe which type of hardware sensor is being represented by a particular Sensor object.

- **Supported Android Sensors**
- ‰ Sensor.TYPE_AMBIENT_TEMPERATURE — Introduced in Android 4.0 (API level 14) to replace the ambiguous — and deprecated — Sensor.TYPE_TEMPERATURE. This is a thermometer that returns the temperature in degrees Celsius; the temperature returned will be the ambient room temperature.

- Sensor.TYPE_ACCELEROMETER — A three-axis accelerometer that returns the current acceleration along three axes in m/s2 (meters per second, per second).

- Sensor.TYPE_GRAVITY — A three-axis gravity sensor that returns the current direction and magnitude of gravity along three axes in m/s2. The gravity sensor typically is implemented as a virtual sensor by applying a low-pass filter to the accelerometer sensor results.

- Sensor.TYPE_GYROSCOPE — A three-axis gyroscope that returns the rate of device rotation along three axes in radians/second. You can integrate the rate of rotation over time to determine the current orientation of the device.

- Sensor.TYPE_MAGNETIC_FIELD — A magnetometer that finds the current magnetic field in microteslas (μT) along three axes.

- Sensor.TYPE_PRESSURE — An atmospheric pressure sensor, or barometer, that returns the current atmospheric pressure in millibars (mbars) as a single value.

- Sensor.TYPE_PROXIMITY — A proximity sensor that indicates the distance between the device and the target object in centimeters.

- Sensor.TYPE_LIGHT — An ambient light sensor that returns a single value describing the ambient illumination in lux. A light sensor commonly is used to control the screen brightness dynamically.

- **Introducing Virtual Sensors**

- **Finding Sensors**
- To find every Sensor available on the host platform, use getSensorList on the Sensor Manager, passing in Sensor.TYPE_ALL:
- List<Sensor> allSensors = sensorManager.getSensorList(Sensor.TYPE_ALL);

- List<Sensor> gyroscopes = sensorManager.getSensorList(Sensor.TYPE_GYROSCOPE);

- You can find the default Sensor implementation for a given type by using the Sensor Manager’s getDefaultSensor method.

- **Monitoring Sensors**
- To monitor a Sensor, implement a SensorEventListener, using the onSensorChanged method to monitor Sensor values, and onAccuracyChanged to react to changes in a Sensor's accuracy.

- The SensorEvent parameter in the onSensorChanged method includes the following four properties to describe each Sensor Event:
- ‰ sensor — The Sensor object that triggered the event.
- ‰ accuracy — The accuracy of the Sensor when the event occurred (low, medium, high, or unreliable).
- ‰ values — A float array that contains the new value(s) observed.
- ‰ timestamp — The time (in nanoseconds) at which the Sensor Event occurred.

- You can monitor changes in the accuracy of a Sensor separately, using the onAccuracyChanged method.
- In both handlers the accuracy value represents the Sensor’s accuracy, using one of the following constants:
- ‰ SensorManager.SENSOR_STATUS_ACCURACY_LOW — Indicates that the Sensor is reporting with low accuracy and needs to be calibrated
- ‰ SensorManager.SENSOR_STATUS_ACCURACY_MEDIUM — Indicates that the Sensor data is of average accuracy and that calibration might improve the accuracy of the reported results
- ‰ SensorManager.SENSOR_STATUS_ACCURACY_HIGH — Indicates that the Sensor is reporting with the highest possible accuracy
- ‰ SensorManager.SENSOR_STATUS_UNRELIABLE — Indicates that the Sensor data is unreliable, meaning that either calibration is required or readings are not currently possible

- To listen for Sensor Events, register your Sensor Event Listener with the Sensor Manager. Specify the Sensor to observe, and the rate at which you want to receive updates.

- The Sensor Manager includes the following static constants (shown in descending order of responsiveness) to let you specify a suitable update rate:
- ‰ SENSOR_DELAY_FASTEST — Specifies the fastest possible update rate
- ‰ SENSOR_DELAY_GAME — Specifies an update rate suitable for use in controlling games
- ‰ SENSOR_DELAY_NORMAL — Specifies the default update rate
- ‰ SENSOR_DELAY_UI — Specifies a rate suitable for updating UI features

- **Interpreting Sensor Values**

SENSOR TYPE	VALUE COUNT	VALUE COMPOSITION	COMMENTARY
`TYPE_ACCELEROMETER`	3	value[0] : X-axis (Lateral) value[1] : Y-axis (Longitudinal) value[2] : Z-axis (Vertical)	Acceleration along three axes in m/s^2. The Sensor Manager includes a set of gravity constants of the form `SensorManager.GRAVITY_*`.
SENSOR TYPE	**VALUE COUNT**	**VALUE COMPOSITION**	**COMMENTARY**
`TYPE_GRAVITY`	3	value[0] : X-axis (Lateral) value[1] : Y-axis (Longitudinal) value[2] : Z-axis (Vertical)	Force of gravity along three axes in m/s^2. The Sensor Manager includes a set of gravity constants of the form `SensorManager.GRAVITY_*`.
`TYPE_HUMIDITY`	1	value[0]: Relative humidity	Relative humidity as a percentage (%).
`TYPE_LINEAR_ACCELERATION`	3	value[0] : X-axis (Lateral) value[1] : Y-axis (Longitudinal) value[2] : Z-axis (Vertical)	Linear acceleration along three axes in m/s^2 without the force of gravity.

`TYPE_GYROSCOPE`	3	value[0] : X-axis value[1] : Y-axis value[2] : Z-axis	Rate of rotation around three axes in radians/second (r/s).
`TYPE_ROTATION_VECTOR`	3 (+1 optional)	values[0]: x*sin(θ/2) values[1]: y*sin(θ/2) values[2]: z*sin(θ/2) values[3]: cos(θ/2) (optional)	Device orientation described as an angle of rotation around an axis (°).
`TYPE_MAGNETIC_FIELD`	3	value[0] : X-axis (Lateral) value[1] : Y-axis (Longitudinal) value[2] : Z-axis (Vertical)	Ambient magnetic field measured in microteslas (μT).
`TYPE_LIGHT`	1	value[0] : Illumination	Ambient light measured in lux (lx). The Sensor Manager includes a set of constants representing different standard illuminations of the form `SensorManager.LIGHT_*`.

SENSOR TYPE	VALUE COUNT	VALUE COMPOSITION	COMMENTARY
`TYPE_PRESSURE`	1	value[0] : Atmospheric Pressure	Atmospheric pressure measured in millibars (mbars).
`TYPE_PROXIMITY`	1	value[0] : Distance	Distance from target measured in centimeters (cm).
`TYPE_AMBIENT_TEMPERATURE`	1	value[0] : Temperature	Ambient temperature measured in degrees Celsius (°C).

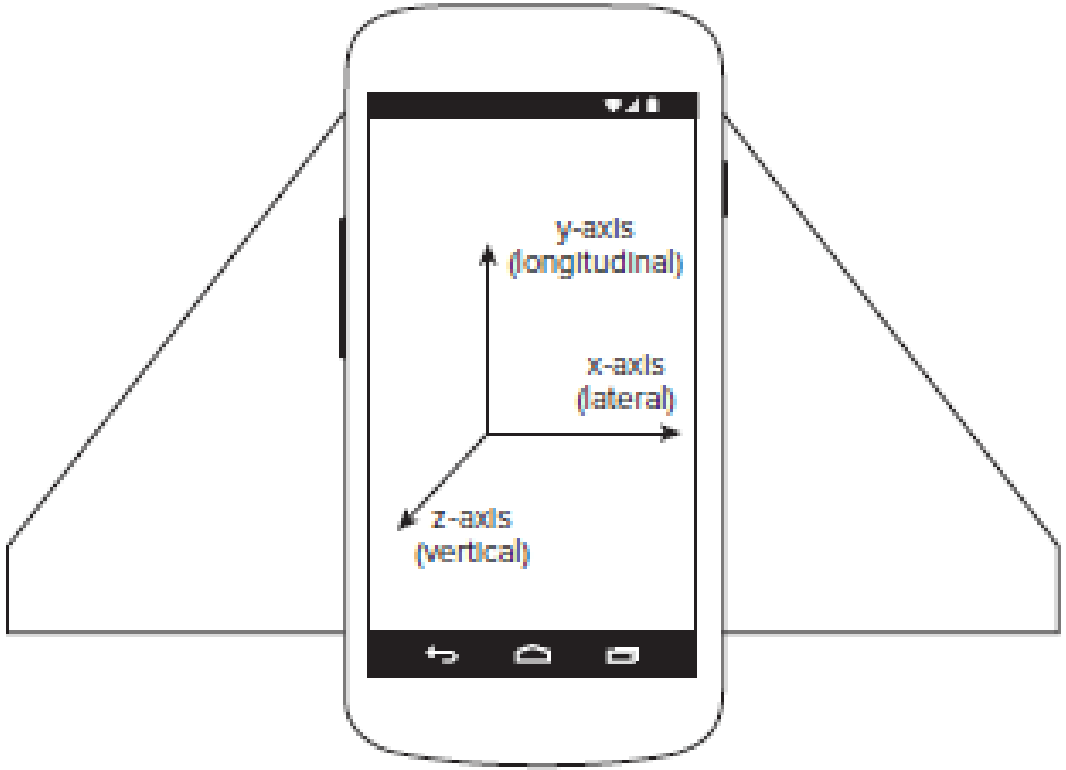
- **MONITORING A DEVICE'S MOVEMENT AND ORIENTATION**

- Where they are available, movement and orientation sensors can be used by your application to:
- ‰ Determine the device orientation
- ‰ React to changes in orientation
- ‰ React to movement or acceleration
- ‰ Understand which direction the user is facing
- ‰ Monitor gestures based on movement, rotation, or acceleration

- This opens some intriguing possibilities for your applications. By monitoring orientation, direction, and movement, you can:
- ‰ Use the compass and accelerometers to determine your heading and orientation. Use these with a map, camera, and location-based service to create augmented-reality UIs that overlay location-based data over a real-time camera feed.
- ‰ Create UIs that adjust dynamically as the orientation of the device changes. In the most simple case, Android alters the screen orientation when the device is rotated from portrait to landscape or vice versa, but applications such as the native Gallery use orientation changes to provide a 3D effect on stacks of photos.
- ‰ Measure movement or vibration. For example, you could create an application that lets a user lock his or her device; if any movement is detected while it’s locked, it could send an alert SMS that includes the current location.
- ‰ Create UI controls that use physical gestures and movement as input.

- **Introducing Accelerometers**
- *Acceleration* is defined as the rate of change of velocity; that means accelerometers measure how quickly the speed of the device is changing in a given direction. Using an accelerometer you can detect movement and, more usefully, the rate of change of the speed of that movement (also known as *linear acceleration*).

- **Detecting Acceleration Changes**
- Acceleration can be measured along three directional axes:
- ‰ Left-right (lateral)
- ‰ Forward-backward (longitudinal)
- ‰ Up-down (vertical)


y-axis
(longitudinal)
x-axis
(lateral)
z-axis
(vertical)

- final SensorEventListener mySensorEventListener = new SensorEventListener() {
- public void onSensorChanged(SensorEvent sensorEvent) {
- if (sensorEvent.sensor.getType() == Sensor.TYPE_ACCELEROMETER) {
- **float xAxis_lateralA = sensorEvent.values[0];**
- **float yAxis_longitudinalA = sensorEvent.values[1];**
- **float zAxis_verticalA = sensorEvent.values[2];**
- // TODO apply the acceleration changes to your application.
- }
- }
- public void onAccuracyChanged(Sensor sensor, int accuracy) {}
- };

- I will refer to the movement of the device in relation to its natural orientation, which may be either landscape or portrait.
- ‰ **x-axis (lateral)** — Sideways (left or right) acceleration, for which positive values represent movement toward the right, and negative values indicate movement to the left.
- ‰ **y-axis (longitudinal)** — Forward or backward acceleration, for which forward acceleration, such as the device being pushed in the direction of the top of the device, is represented by a positive value and acceleration backwards represented by negative values.
- ‰ **z-axis (vertical)** — Upward or downward acceleration, for which positive represents upward movement, such as the device being lifted. While at rest at the device's natural orientation, the vertical accelerometer will register –9.8m/s2 as a result of gravity.

- Telephony and SMS
- Android also offers full access to SMS functionality, letting you send and receive SMS messages from within your applications. Using the Android APIs, you can create your own SMS client application to replace the native clients available as part of the software stack.

- **HARDWARE SUPPORT FOR TELEPHONY**
- With the arrival of Wi-Fi-only Android devices, you can no longer assume that telephony will be supported on all the hardware on which your application may be available.

- **Marking Telephony as a Required Hardware Feature**
- To specify that your application requires telephony support to function, you can add a uses-feature node to your application manifest:
- <uses-feature android:name=”android.hardware.telephony”
- android:required=”true”/>

- **Checking for Telephony Hardware**
- Use the Package Manager’s hasSystemFeature method, specifying the FEATURE_TELEPHONY feature.
- The Package Manager also includes constants to query the existence of CDMA- and GSM-specific hardware.
- PackageManager pm = getPackageManager();
- boolean telephonySupported =
- pm.hasSystemFeature(PackageManager.FEATURE_TELEPHONY);
- boolean gsmSupported =
- pm.hasSystemFeature(PackageManager.FEATURE_TELEPHONY_CDMA);
- boolean cdmaSupported =
- pm.hasSystemFeature(PackageManager.FEATURE_TELEPHONY_GSM);

- **USING TELEPHONY**
- **Initiating Phone Calls**
- Best practice for initiating phone calls is to use an Intent.ACTION_DIAL Intent, specifying the number to dial by setting the Intents data using a tel: schema:
- Intent whoyougonnacall = new Intent(Intent.ACTION_DIAL,
- Uri.parse(“tel:555-2368”));
- startActivity(whoyougonnacall);
- This starts a dialer Activity that should be prepopulated with the number you specifi ed. The default dialer Activity allows the user to change the number before explicitly initiating the call. As a result,
- using the ACTION_DIAL Intent action doesn’t require any special permissions.

- **Accessing Telephony Properties and Phone State**
- Access to the telephony APIs is managed by the Telephony Manager, accessible using the getSystemService method:
- String srvcName = Context.TELEPHONY_SERVICE;
- TelephonyManager telephonyManager =
- (TelephonyManager)getSystemService(srvcName);
- The Telephony Manager provides direct access to many of the phone properties, including device, network, subscriber identity module (SIM), and data state details. You can also access some connectivity
- status information, although this is usually done using the Connectivity Manager

- Reading Phone Device Details
- Using the Telephony Manager, you can obtain the phone type (GSM CDMA, or SIP), unique ID (IMEI or MEID), software version, and the phone's phone number:
- String phoneTypeStr = "unknown";
- int phoneType = telephonyManager.getPhoneType();
- switch (phoneType) {
- case (TelephonyManager.PHONE_TYPE_CDMA): phoneTypeStr = "CDMA";
- break;
- case (TelephonyManager.PHONE_TYPE_GSM) : phoneTypeStr = "GSM";
- break;
- case (TelephonyManager.PHONE_TYPE_SIP): phoneTypeStr = "SIP";
- break;
- case (TelephonyManager.PHONE_TYPE_NONE): phoneTypeStr = "None";
- break;
- default: break;
- }

- Reading SIM Details
- If your application is running on a GSM device, it will usually have a SIM. You can query the SIM details from the Telephony Manager to obtain the ISO country code, operator name, and operator
- MCC and MNC for the SIM installed in the current device. These details can be useful if you need to provide specialized functionality for a particular carrier.
- If you have included the READ_PHONE_STATE uses-permission in your application manifest, you can also obtain the serial number for the current SIM using the getSimSerialNumber method when the SIM is in a ready state.

- Before you can use any of these methods, you must ensure that the SIM is in a ready state. You can determine this using the getSimState method:
- int simState = telephonyManager.getSimState();
- switch (simState) {
- case (TelephonyManager.SIM_STATE_ABSENT): break;
- case (TelephonyManager.SIM_STATE_NETWORK_LOCKED): break;
- case (TelephonyManager.SIM_STATE_PIN_REQUIRED): break;
- case (TelephonyManager.SIM_STATE_PUK_REQUIRED): break;
- case (TelephonyManager.SIM_STATE_UNKNOWN): break;
- case (TelephonyManager.SIM_STATE_READY): {

- **Monitoring Changes in Phone State Using the Phone State Listener**
- The Android telephony APIs lets you monitor changes to phone state and associated details such as incoming phone numbers.
- Changes to the phone state are monitored using the PhoneStateListener class, with some state changes also broadcast as Intents. This section describes how to use the Phone State Listener, and
- the following section describes which Broadcast Intents are available.
- To monitor and manage phone state, your application must specify the READ_PHONE_STATE
- uses-permission:
- <uses-permission android:name=”android.permission.READ_PHONE_STATE”/>

- **Using Intent Receivers to Monitor Incoming Phone Calls**
- When the phone state changes as a result of an incoming, accepted, or terminated phone call, the Telephony Manager will broadcast an ACTION_PHONE_STATE_CHANGED Intent.
- By registering a manifest Intent Receiver that listens for this Broadcast Intent, as shown in the snippet below, you can listen for incoming phone calls at any time, even if your application isn't
- running. Note that your application needs to request the READ_PHONE_STATE permission to receive the phone state changed Broadcast Intent.
- <receiver android:name="PhoneStateChangedReceiver">
- <intent-filter>
- **<action android:name="android.intent.action.PHONE_STATE"></action>**
- </intent-filter>
- </receiver>

- **INTRODUCING SMS**
- If you own a mobile phone that’s less than two decades old, chances are you’re familiar with SMS messaging. SMS is now one of the most-used communication mechanisms on mobile phones.
- SMS technology is designed to send short text messages between mobile phones. It provides support for sending both text messages (designed to be read by people) and data messages (meant to be
- consumed by applications). Multimedia messaging service (MMS) messages allow users to send and receive messages that include multimedia attachments such as photos, videos, and audio.

- **Using SMS**
- Android provides support for sending both SMS and MMS messages using a messaging application installed on the device with the SEND and SEND_TO Broadcast Intents.
- Android also supports full SMS functionality within your applications through the SmsManager class. Using the SMS Manager, you can replace the native SMS application to send text messages,
- react to incoming texts, or use SMS as a data transport layer.

- **Sending SMS**
- In most cases it’s best practice to use an Intent to send SMS and MMS messages using another application — typically the native SMS application — rather than implementing a full SMS client.
- To do so, call startActivity with an Intent.ACTION_SENDTO action Intent. Specify a target number using sms: schema notation as the Intent data. Include the message you want to send within
- the Intent payload using an sms_body extra:
- Intent smsIntent = new Intent(Intent.ACTION_SENDTO,
- Uri.parse(“sms:55512345”));
- smsIntent.putExtra(“sms_body”, “Press send to send me”);
- startActivity(smsIntent);

- To attach fi les to your message (effectively creating an MMS message), add an Intent.EXTRA_STREAM with the URI of the resource to attach, and set the Intent type to the MIME type of the attached resource.
- Note that the native MMS application doesn't include an Intent Receiver for ACTION_SENDTO with a type set. Instead, you need to use ACTION_SEND and include the target phone number as an address extra:
- // Get the URI of a piece of media to attach.
- Uri attached_Uri
- = Uri.parse("content://media/external/images/media/1");
- // Create a new MMS intent
- Intent mmsIntent = new Intent(Intent.ACTION_SEND, attached_Uri);
- mmsIntent.putExtra("sms_body", "Please see the attached image");
- mmsIntent.putExtra("address", "07912355432");
- mmsIntent.putExtra(Intent.EXTRA_STREAM, attached_Uri);
- mmsIntent.setType("image/jpeg");
- startActivity(mmsIntent);

- **Sending SMS Messages Using the SMS Manager**
- SMS messaging in Android is handled by the SmsManager class. You can get a reference to the SMS Manager using the static SmsManager.getDefault method:
- SmsManager smsManager = SmsManager.getDefault();
- To send SMS messages, your application must specify the SEND_SMS uses-permission:
- <uses-permission android:name=”android.permission.SEND_SMS”/>

- Sending Text Messages
- To send a text message, use sendTextMessage from the SMS Manager, passing in the address (phone number) of your recipient and the text message you want to send:
- SmsManager smsManager = SmsManager.getDefault();
- String sendTo = “5551234”;
- String myMessage = “Android supports programmatic SMS messaging!”;
- **smsManager.sendTextMessage(sendTo, null, myMessage, null, null);**

- **Listening for Incoming SMS Messages**
- When a device receives a new SMS message, a new Broadcast Intent is fi red with the android.provider.
- Telephony.SMS_RECEIVED action. Note that this is a string literal; the SDK currently doesn’t include a reference to this string, so you must specify it explicitly when using it in your applications.
- For an application to listen for SMS Broadcast Intents, it needs to specify the RECEIVE_SMS manifest
- permission:
- <uses-permission
- android:name=”android.permission.RECEIVE_SMS”
- />

- objects packaged within the SMS Broadcast Intent bundle, use the pdu key to extract an array of SMS PDUs (protocol data units — used to encapsulate an SMS message and its metadata), each of which represents an SMS message, from the extras Bundle. To convert each PDU byte array into an SMS Message object, call SmsMessage.createFromPdu, passing in each byte array:
- Bundle bundle = intent.getExtras();
- if (bundle != null) {
- Object[] pdus = (Object[]) bundle.get(“pdus”);
- SmsMessage[] messages = new SmsMessage[pdus.length];
- for (int i = 0; i < pdus.length; i++)
- messages[i] = SmsMessage.createFromPdu((byte[]) pdus[i]);

- Each SmsMessage contains the SMS message details, including the originating address (phone number),
- timestamp, and the message body, which can be extracted using the getOriginatingAddress,
- getTimestampMillis, and getMessageBody methods, respectively:
- public class MySMSReceiver extends BroadcastReceiver {
- @Override
- public void onReceive(Context context, Intent intent) {
- Bundle bundle = intent.getExtras();
- if (bundle != null) {
- Object[] pdus = (Object[]) bundle.get(“pdus”);
- SmsMessage[] messages = new SmsMessage[pdus.length];
- for (int i = 0; i < pdus.length; i++)
- messages[i] = SmsMessage.createFromPdu((byte[]) pdus[i]);
- **for (SmsMessage message : messages) {**
- **String msg = message.getMessageBody();**
- **long when = message.getTimestampMillis();**
- **String from = message.getOriginatingAddress();**
- **Toast.makeText(context, from + “ : “ + msg,**
- **Toast.LENGTH_LONG).show();**

- Thank you