- Getting Started
  - Building Your First App
    - Creating an Android Project
    - Running Your Application
    - Building a Simple User Interface
    - Starting Another Activity
  - Adding the Action Bar
    - Setting Up the Action Bar
    - Adding Action Buttons
    - Styling the Action Bar
    - Overlaying the Action Bar
  - Supporting Different Devices
    - Supporting Different Languages
    - Supporting Different Screens
    - Supporting Different Platform Versions
  - Managing the Activity Lifecycle
    - Starting an Activity
    - Pausing and Resuming an Activity
    - Stopping and Restarting an Activity
    - Recreating an Activity
  - Building a Dynamic UI with Fragments
    - Creating a Fragment
    - Building a Flexible UI
    - Communicating with Other Fragments
  - Saving Data
    - Saving Key-Value Sets
    - Saving Files
    - Saving Data in SQL Databases
  - Interacting with Other Apps
    - Sending the User to Another App
    - Getting a Result from the Activity
    - Allowing Other Apps to Start Your Activity
- Building Apps with Content Sharing
  - Sharing Simple Data
    - Sending Simple Data to Other Apps
    - Receiving Simple Data from Other Apps

        - Adding an Easy Share Action
    - Sharing Files
        - Setting Up File Sharing
        - Sharing a File
        - Requesting a Shared File
        - Retrieving File Information
    - Sharing Files with NFC
        - Sending Files to Another Device
        - Receiving Files from Another Device
- Building Apps with Multimedia
    - Managing Audio Playback
        - Controlling Your App's Volume and Playback
        - Managing Audio Focus
        - Dealing with Audio Output Hardware
    - Capturing Photos
        - Taking Photos Simply
        - Recording Videos Simply
        - Controlling the Camera
    - Printing Content
        - Photos
        - HTML Documents
        - Custom Documents
- Building Apps with Graphics & Animation
    - Displaying Bitmaps Efficiently
        - Loading Large Bitmaps Efficiently
        - Processing Bitmaps Off the UI Thread
        - Caching Bitmaps
        - Managing Bitmap Memory
        - Displaying Bitmaps in Your UI
    - Displaying Graphics with OpenGL ES
        - Building an OpenGL ES Environment
        - Defining Shapes
        - Drawing Shapes
        - Applying Projection and Camera Views
        - Adding Motion

        - Responding to Touch Events
    - Animating Views Using Scenes and Transitions
        - The Transitions Framework
        - Creating a Scene
        - Applying a Transition
        - Creating Custom Transitions
    - Adding Animations
        - Crossfading Two Views
        - Using ViewPager for Screen Slide
        - Displaying Card Flip Animations
        - Zooming a View
        - Animating Layout Changes
- Building Apps with Connectivity & the Cloud
    - Connecting Devices Wirelessly
        - Using Network Service Discovery
        - Creating P2P Connections with Wi-Fi
        - Using Wi-Fi P2P for Service Discovery
    - Performing Network Operations
        - Connecting to the Network
        - Managing Network Usage
        - Parsing XML Data
    - Transferring Data Without Draining the Battery
        - Optimizing Downloads for Efficient Network Access
        - Minimizing the Effect of Regular Updates
        - Redundant Downloads are Redundant
        - Modifying Patterns Based on the Connectivity Type
    - Syncing to the Cloud
        - Using the Backup API
    - Resolving Cloud Save Conflicts
    - Transferring Data Using Sync Adapters
        - Creating a Stub Authenticator
        - Creating a Stub Content Provider
        - Creating a Sync Adapter
        - Running a Sync Adapter
    - Transmitting Network Data Using Volley
        - Sending a Simple Request

        - Setting Up a RequestQueue
        - Making a Standard Request
        - Implementing a Custom Request
- Building Apps with Location & Maps
    - Making Your App Location-Aware
        - Getting the Last Known Location
        - Receiving Location Updates
        - Displaying a Location Address
        - Creating and Monitoring Geofences
    - Adding Maps
- Building Apps with User Info & Sign-In
    - Accessing Contacts Data
        - Retrieving a List of Contacts
        - Retrieving Details for a Contact
        - Modifying Contacts Using Intents
        - Displaying the Quick Contact Badge
    - Adding Sign-In
- Building Apps for Wearables
    - Adding Wearable Features to Notifications
        - Creating a Notification
        - Receiving Voice Input in a Notification
        - Adding Pages to a Notification
        - Stacking Notifications
    - Creating Wearable Apps
        - Creating and Running a Wearable App
        - Creating Custom Layouts
        - Keeping Your App Visible
        - Adding Voice Capabilities
        - Packaging Wearable Apps
        - Debugging over Bluetooth
    - Creating Custom UIs
        - Defining Layouts
        - Creating Cards
        - Creating Lists

        - Creating a 2D Picker
        - Showing Confirmations
        - Exiting Full-Screen Activities
    - Sending and Syncing Data
        - Accessing the Wearable Data Layer
        - Syncing Data Items
        - Transferring Assets
        - Sending and Receiving Messages
        - Handling Data Layer Events
    - Creating Watch Faces
        - Designing Watch Faces
        - Building a Watch Face Service
        - Drawing Watch Faces
        - Showing Information in Watch Faces
        - Providing Configuration Activities
        - Addressing Common Issues
        - Optimizing Performance and Battery Life
    - Detecting Location
- Building Apps for TV
    - Building TV Apps
        - Getting Started with TV Apps
        - Handling TV Hardware
        - Building TV Layouts
        - Creating TV Navigation
    - Building TV Playback Apps
        - Creating a Catalog Browser
        - Providing a Card View
        - Building a Details View
        - Displaying a Now Playing Card
    - Helping Users Find Content on TV
        - Recommending TV Content
        - Making TV Apps Searchable
        - Searching within TV Apps
    - Building TV Games
    - Building TV Channels
        - Developing a TV Input Service

        - Working with Channel Data
        - Managing User Interaction
    - TV Apps Checklist
- Building Apps for Auto
    - Getting Started with Auto
    - Playing Audio for Auto
    - Messaging for Auto
- Building Apps for Work
    - Ensuring Compatibility with Managed Profiles
    - Implementing App Restrictions
    - Building a Work Policy Controller
- Best Practices for Interaction & Engagement
    - Designing Effective Navigation
        - Planning Screens and Their Relationships
        - Planning for Multiple Touchscreen Sizes
        - Providing Descendant and Lateral Navigation
        - Providing Ancestral and Temporal Navigation
        - Putting it All Together: Wireframing the Example App
    - Implementing Effective Navigation
        - Creating Swipe Views with Tabs
        - Creating a Navigation Drawer
        - Providing Up Navigation
        - Providing Proper Back Navigation
        - Implementing Descendant Navigation
    - Notifying the User
        - Building a Notification
        - Preserving Navigation when Starting an Activity
        - Updating Notifications
        - Using Big View Styles
        - Displaying Progress in a Notification
    - Supporting Swipe-to-Refresh
        - Adding Swipe-to-Refresh To Your App
        - Responding to a Refresh Gesture
    - Adding Search Functionality

        - Setting up the Search Interface
        - Storing and Searching for Data
        - Remaining Backward Compatible
    - Making Your App Content Searchable by Google
        - Enabling Deep Links for App Content
        - Specifying App Content for Indexing
- Best Practices for User Interface
    - Designing for Multiple Screens
        - Supporting Different Screen Sizes
        - Supporting Different Screen Densities
        - Implementing Adaptive UI Flows
    - Creating Custom Views
        - Creating a Custom View Class
        - Implementing Custom Drawing
        - Making the View Interactive
        - Optimizing the View
    - Creating Backward-Compatible UIs
        - Abstracting the New APIs
        - Proxying to the New APIs
        - Creating an Implementation with Older APIs
        - Using the Version-Aware Component
    - Implementing Accessibility
        - Developing Accessible Applications
        - Developing Accessibility Services
    - Managing the System UI
        - Dimming the System Bars
        - Hiding the Status Bar
        - Hiding the Navigation Bar
        - Using Immersive Full-Screen Mode
        - Responding to UI Visibility Changes
    - Creating Apps with Material Design
        - Getting Started
        - Using the Material Theme
        - Creating Lists and Cards
        - Defining Shadows and Clipping Views
        - Working with Drawables

        - Defining Custom Animations
        - Maintaining Compatibility
- Best Practices for User Input
    - Using Touch Gestures
        - Detecting Common Gestures
        - Tracking Movement
        - Animating a Scroll Gesture
        - Handling Multi-Touch Gestures
        - Dragging and Scaling
        - Managing Touch Events in a ViewGroup
    - Handling Keyboard Input
        - Specifying the Input Method Type
        - Handling Input Method Visibility
        - Supporting Keyboard Navigation
        - Handling Keyboard Actions
    - Supporting Game Controllers
        - Handling Controller Actions
        - Supporting Controllers Across Android Versions
        - Supporting Multiple Game Controllers
- Best Practices for Background Jobs
    - Running in a Background Service
        - Creating a Background Service
        - Sending Work Requests to the Background Service
        - Reporting Work Status
    - Loading Data in the Background
        - Running a Query with a CursorLoader
        - Handling the Results
    - Managing Device Awake State
        - Keeping the Device Awake
        - Scheduling Repeating Alarms
- Best Practices for Performance
    - Managing Your App's Memory
    - Performance Tips
    - Improving Layout Performance

        - Optimizing Layout Hierarchies
        - Re-using Layouts with <include/>
        - Loading Views On Demand
        - Making ListView Scrolling Smooth
    - Optimizing Battery Life
        - Monitoring the Battery Level and Charging State
        - Determining and Monitoring the Docking State and Type
        - Determining and Monitoring the Connectivity Status
        - Manipulating Broadcast Receivers On Demand
    - Sending Operations to Multiple Threads
        - Specifying the Code to Run on a Thread
        - Creating a Manager for Multiple Threads
        - Running Code on a Thread Pool Thread
        - Communicating with the UI Thread
    - Keeping Your App Responsive
    - JNI Tips
    - SMP Primer for Android
- Best Practices for Security & Privacy
    - Security Tips
    - Security with HTTPS and SSL
    - Updating Your Security Provider to Protect Against SSL Exploits
    - Checking Device Compatibility with SafetyNet
    - Enhancing Security with Device Management Policies
- Best Practices for Testing
    - Testing Your Activity
        - Setting Up Your Test Environment
        - Creating and Running a Test Case
        - Testing UI Components
        - Creating Unit Tests
        - Creating Functional Tests
    - Automating UI Tests
        - Testing UI for a Single App
        - Testing UI for Multiple Apps
    - Building Effective Unit Tests
        - Building Local Unit Tests

        - Building Instrumented Unit Tests
- Using Google Play to Distribute & Monetize
    - Selling In-app Products
        - Preparing Your App
        - Establishing Products for Sale
        - Purchasing Products
        - Testing Your App
    - Maintaining Multiple APKs
        - Creating Multiple APKs for Different API Levels
        - Creating Multiple APKs for Different Screen Sizes
        - Creating Multiple APKs for Different GL Textures
        - Creating Multiple APKs with 2+ Dimensions

Getting Started

Welcome to Training for Android developers. Here you'll find sets of lessons within classes that describe how to accomplish a specific task with code samples you can re-use in your app. Classes are organized into several groups you can see at the top-level of the left navigation.

This first group, *Getting Started*, teaches you the bare essentials for Android app development. If you're a new Android app developer, you should complete each of these classes in order.

If you prefer to learn through interactive video training, check out this trailer for a course about the fundamentals of Android development.

Start the video course

1. **Building Your First App**

   After you've installed the Android SDK, start with this class to learn the basics about Android app development.

   1. Creating an Android Project
   2. Running Your Application
   3. Building a Simple User Interface
   4. Starting Another Activity

2. # Adding the Action Bar

The action bar is one of the most important design elements you can implement for your app's activities. Although first introduced with API level 11, you can use the Support Library to include the action bar on devices running Android 2.1 or higher.

1. Setting Up the Action Bar
2. Adding Action Buttons
3. Styling the Action Bar
4. Overlaying the Action Bar

3. # Supporting Different Devices

How to build your app with alternative resources that provide an optimized user experience on multiple device form factors using a single APK.

1. Supporting Different Languages
2. Supporting Different Screens
3. Supporting Different Platform Versions

4. # Managing the Activity Lifecycle

How Android activities live and die and how to create a seamless user experience by implementing lifecycle callback methods.

1. Starting an Activity
2. Pausing and Resuming an Activity
3. Stopping and Restarting an Activity
4. Recreating an Activity

5. # Building a Dynamic UI with Fragments

How to build a user interface for your app that is flexible enough to present multiple UI components on large screens and a more

constrained set of UI components on smaller screens—essential for building a single APK for both phones and tablets.

1. Creating a Fragment
2. Building a Flexible UI
3. Communicating with Other Fragments

6. Saving Data

How to save data on the device, whether it's temporary files, downloaded app assets, user media, structured data, or something else.

1. Saving Key-Value Sets
2. Saving Files
3. Saving Data in SQL Databases

7. Interacting with Other Apps

How to build a user experience that leverages other apps available on the device to perform advanced user tasks, such as capture a photo or view an address on a map.

1. Sending the User to Another App
2. Getting a Result from the Activity
3. Allowing Other Apps to Start Your Activity

Online video courses

If you prefer to learn through interactive video training, check out these free courses.

Building Your First App

Previous Next Get started

Dependencies and prerequisites

- Android Studio

Welcome to Android application development!

This class teaches you how to build your first Android app. You'll learn how to create an Android project and run a debuggable version of the app. You'll also learn some fundamentals of Android app design, including how to build a simple user interface and handle user input.

Set Up Your Environment

Before you start this class, be sure you have your development environment set up. You need to:

1. Download Android Studio.
2. Download the latest SDK tools and platforms using the SDK Manager.

Note: Make sure you install the most recent versions of Android Studio and the Android SDK before you start this class. The procedures described in this class may not apply to earlier versions.

If you haven't already done these tasks, start by downloading the Android SDK and following the install steps. Once you've finished the setup, you're ready to begin this class.

This class uses a tutorial format that incrementally builds a small Android app that teaches you some fundamental concepts about Android

development, so it's important that you follow each step.

Next Get started Next class

Creating an Android Project

Previous Next Get started

This lesson teaches you to

1. Create a Project with Android Studio
2. Create a Project with Command Line Tools

You should also read

- Installing the SDK
- Managing Projects

An Android project contains all the files that comprise the source code for your Android app. The Android SDK tools make it easy to start a new Android project with a set of default project directories and files.

This lesson shows how to create a new project either using Android Studio or using the SDK tools from a command line.

Note: You should already have the Android SDK installed, and if you're using Android Studio, you should also have Android Studio installed. If you don't have these, follow the guide to Installing the Android SDK before you start this lesson.

Create a Project with Android Studio

1. In Android Studio, create a new project:
    - If you don't have a project opened, in the **Welcome** screen, click **New Project**.
    - If you have a project opened, from the **File** menu, select **New Project**.

Figure 1. Configuring a new project in Android Studio.

2. Under **Configure your new project**, fill in the fields as shown in figure 1 and click **Next**.

   It will probably be easier to follow these lessons if you use the same values as shown.

   - **Application Name** is the app name that appears to users. For this project, use "My First App."
   - **Company domain** provides a qualifier that will be appended to the package name; Android Studio will remember this qualifier for each new project you create.
   - **Package name** is the fully qualified name for the project (following the same rules as those for naming packages in the Java programming

language). Your package name must be unique across all packages installed on the Android system. You can **Edit** this value independently from the application name or the company domain.
    - **Project location** is the directory on your system that holds the project files.
3. Under **Select the form factors your app will run on**, check the box for **Phone and Tablet**.
4. For **Minimum SDK**, select **API 8: Android 2.2 (Froyo)**.

   The Minimum Required SDK is the earliest version of Android that your app supports, indicated using the API level. To support as many devices as possible, you should set this to the lowest version available that allows your app to provide its core feature set. If any feature of your app is possible only on newer versions of Android and it's not critical to the app's core feature set, you can enable the feature only when running on the versions that support it (as discussed in Supporting Different Platform Versions).

5. Leave all of the other options (TV, Wear, and Glass) unchecked and click **Next.**

   ### Activities

   An activity is one of the distinguishing features of the Android framework. Activities provide the user with access to your app, and there may be many activities. An application will usually have a main activity for when the user launches the application, another activity for when she selects some content to view, for example, and other activities for when she performs other tasks within the app. See Activities for more information.

6. Under **Add an activity to <***template***>**, select **Blank Activity** and click **Next**.
7. Under **Choose options for your new file**, change the **Activity Name** to *MyActivity*. The **Layout Name** changes to *activity_my*, and the **Title** to *MyActivity*. The **Menu Resource Name** is *menu_my*.
8. Click the **Finish** button to create the project.

Your Android project is now a basic "Hello World" app that contains some default files. Take a moment to review the most important of these:

`app/src/main/res/layout/activity_my.xml`
: This is the XML layout file for the activity you added when you created the project with Android Studio. Following the New Project workflow, Android Studio presents this file with both a text view and a preview of the screen UI. The file includes some default settings and a `TextView` element that displays the message, "Hello world!"

`app/src/main/java/com.mycompany.myfirstapp/MyActivity.java`

A tab for this file appears in Android Studio when the New Project workflow finishes. When you select the file you see the class definition for the activity you created. When you build and run the app, the `Activity` class starts the activity and loads the layout file that says "Hello World!"

`app/src/main/AndroidManifest.xml`

The manifest file describes the fundamental characteristics of the app and defines each of its components. You'll revisit this file as you follow these lessons and add more components to your app.

`app/build.gradle`

Android Studio uses Gradle to compile and build your app. There is a `build.gradle` file for each module of your project, as well as a `build.gradle` file for the entire project. Usually, you're only interested in the `build.gradle` file for the module, in this case the `app` or application module. This is where your app's build dependencies are set, including the `defaultConfig` settings:

- `compiledSdkVersion` is the platform version against which you will compile your app. By default, this is set to the latest version of Android available in your SDK. (It should be Android 4.1 or greater; if you don't have such a version available, you must install one using the SDK Manager.) You can still build your app to support older versions, but setting this to the latest version allows you to enable new features and optimize your app for a great user experience on the latest devices.
- `applicationId` is the fully qualified package name for your application that you specified during the New Project workflow.
- `minSdkVersion` is the Minimum SDK version you specified during the New Project workflow. This is the earliest version of the Android SDK that your app supports.
- `targetSdkVersion` indicates the highest version of Android with which you have tested your application. As new versions of Android become available, you should test your app on the new version and update this value to match the latest API level and thereby take advantage of new platform features. For more information, read Supporting Different Platform Versions.

See Building Your Project with Gradle for more information about Gradle.

Note also the `/res` subdirectories that contain the resources for your application:

`drawable<density>/`

Directories for drawable objects (such as bitmaps) that are designed for various densities, such as medium-density (mdpi) and high-density (hdpi) screens. Other drawable directories contain assets designed for other screen

densities. Here you'll find the ic_launcher.png that appears when you run the default app.

`layout/`
: Directory for files that define your app's user interface like activity_my.xml, discussed above, which describes a basic layout for the MyActivity class.

`menu/`
: Directory for files that define your app's menu items.

`values/`
: Directory for other XML files that contain a collection of resources, such as string and color definitions. The strings.xml file defines the "Hello world!" string that displays when you run the default app.

To run the app, continue to the next lesson.

Create a Project with Command Line Tools

If you're not using the Android Studio IDE, you can instead create your project using the SDK tools from a command line:

1. Change directories into the Android SDK's `sdk/` path.
2. Execute:

   ```
   tools/android list targets
   ```

   This prints a list of the available Android platforms that you've downloaded for your SDK. Find the platform against which you want to compile your app. Make a note of the target ID. We recommend that you select the highest version possible. You can still build your app to support older versions, but setting the build target to the latest version allows you to optimize your app for the latest devices.

   If you don't see any targets listed, you need to install some using the Android SDK Manager tool. See Adding SDK Packages.

3. Execute:

   ```
   android create project --target <target-id> --name MyFirstApp \
   --path <path-to-workspace>/MyFirstApp --activity MyActivity \
   --package com.example.myfirstapp
   ```

   Replace `<target-id>` with an ID from the list of targets (from the previous step) and replace `<path-to-workspace>` with the location in which you want to save your Android projects.

Tip: Add the `platform-tools/` as well as the `tools/` directory to your `PATH`

environment variable.

Your Android project is now a basic "Hello World" app that contains some default files. To run the app, continue to the next lesson.

Next Get started Next class

Running Your App

Previous Next Get started

This lesson teaches you to

1. Run on a Real Device
2. Run on the Emulator

You should also read

- Using Hardware Devices
- Managing Virtual Devices
- Managing Projects

If you followed the previous lesson to create an Android project, it includes a default set of "Hello World" source files that allow you to immediately run the app.

How you run your app depends on two things: whether you have a real device running Android and whether you're using Android Studio. This lesson shows you how to install and run your app on a real device and on the Android emulator, and in both cases with either Android Studio or the command line tools.

Run on a Real Device

If you have a device running Android, here's how to install and run your app.

Set up your device

1. Plug in your device to your development machine with a USB cable. If you're developing on Windows, you might need to install the

appropriate USB driver for your device. For help installing drivers, see the OEM USB Drivers document.

2. Enable **USB debugging** on your device.
   - On most devices running Android 3.2 or older, you can find the option under **Settings > Applications > Development**.
   - On Android 4.0 and newer, it's in **Settings > Developer options**.

     Note: On Android 4.2 and newer, **Developer options** is hidden by default. To make it available, go to **Settings > About phone** and tap **Build number** seven times. Return to the previous screen to find **Developer options**.

Run the app from Android Studio

1. Select one of your project's files and click **Run** from the toolbar.
2. In the **Choose Device** window that appears, select the **Choose a running device** radio button, select your device, and click **OK** .

Android Studio installs the app on your connected device and starts it.

Run the app from a command line

Open a command-line and navigate to the root of your project directory. Use Gradle to build your project in debug mode, invoke the `assembleDebug` build task using the Gradle wrapper script (`gradlew assembleRelease`).

This creates your debug `.apk` file inside the module `build/` directory, named `MyFirstApp-debug.apk`.

On Windows platforms, type this command:

```
> gradlew.bat assembleDebug
```

On Mac OS and Linux platforms, type these commands:

```
$ chmod +x gradlew
$ ./gradlew assembleDebug
```

After you build the project, the output APK for the app module is located in `app/build/outputs/apk/`

Note: The first command (`chmod`) adds the execution permission to the Gradle wrapper script and is only necessary the first time you build this project from the command line.

Make sure the Android SDK `platform-tools/` directory is included in your `PATH` environment variable, then execute:

```
adb install app/build/outputs/MyFirstApp-debug.apk
```

On your device, locate *MyFirstApp* and open it.

That's how you build and run your Android app on a device! To start developing, continue to the next lesson.

Run on the Emulator

Whether you're using Android Studio or the command line, to run your app on the emulator you need to first create an Android Virtual Device (AVD). An AVD is a device configuration for the Android emulator that allows you to model a specific device.

Figure 1. The AVD Manager showing a virtual device.

Create an AVD

1. Launch the Android Virtual Device Manager:
   - In Android Studio, select **Tools > Android > AVD Manager**, or click the AVD Manager icon in the toolbar.
   - Or, from the command line, change directories to `sdk/` and execute:

     ```
     tools/android avd
     ```

     Note: The AVD Manager that appears when launched from the command line is different from the version in Android Studio, so the following instructions may not all apply.

Figure 1. The AVD Manager main screen shows your current virtual devices.

2. On the AVD Manager main screen (figure 1), click **Create Virtual Device**.
3. In the Select Hardware window, select a device configuration, such as Nexus 6, then click **Next**.
4. Select the desired system version for the AVD and click **Next**.
5. Verify the configuration settings, then click **Finish**.

6. In the **Android Virtual Device Manager** window, click **Create**.
7. Enter an **AVD Name**.
8. Select a **Device** type.

   When you select a device type, most of the fields auto-populate.

9. For **Skin** select **HVGA**.
10. For **SD Card**, enter something small, like 10 MiB.

    It really doesn't matter what you enter here since you're not using any storage. But if you reuse this AVD, you might have to adjust this setting.

11. Ignore the **Emulation Options** and click **OK**.
12. In the **Result** screen, click **OK**.
13. Close the **Android Virtual Device Manager** window.

Run the app from Android Studio

1. In **Android Studio**, select your project and click **Run** from the toolbar.
2. In the **Choose Device** window, click the **Launch emulator** radio button.
3. From the **Android virtual device** pull-down menu, select the emulator you created, and click **OK**.

It can take a few minutes for the emulator to load itself. You may have to unlock the screen. When you do, *My First App* appears on the emulator screen.

Run your app from the command line

1. Build the project from the command line. The output APK for the app module is located in `app/build/outputs/apk/`.
2. Make sure the Android SDK `platform-tools/` directory is included in your `PATH` environment variable.
3. Execute this command:

   ```
   adb install app/build/outputs/MyFirstApp-debug.apk
   ```

4. On the emulator, locate *MyFirstApp* and open it.

That's how you build and run your Android app on the emulator! To start developing, continue to the next lesson.

Next Get started Next class

Building a Simple User Interface

Previous Next Get started

This lesson teaches you to

1. Create a Linear Layout
2. Add a Text Field
3. Add String Resources
4. Add a Button
5. Make the Input Box Fill in the Screen Width

You should also read

- Layouts

In this lesson, you create a layout in XML that includes a text field and a button. In the next lesson, your app responds when the button is pressed by sending the content of the text field to another activity.

The graphical user interface for an Android app is built using a hierarchy of `View` and `ViewGroup` objects. `View` objects are usually UI widgets such as buttons or text fields. `ViewGroup` objects are invisible view containers that define how the child views are laid out, such as in a grid or a vertical list.

Android provides an XML vocabulary that corresponds to the subclasses of `View` and `ViewGroup` so you can define your UI in XML using a hierarchy of UI elements.

Layouts are subclasses of the `ViewGroup`. In this exercise, you'll work with a `LinearLayout`.

Alternative Layouts

Declaring your UI layout in XML rather than runtime code is useful for several reasons, but it's especially important so you can create different layouts for different screen sizes. For example, you can create two versions of a layout and tell the system to use one on "small" screens and the other on "large" screens. For more information, see the class about Supporting Different Devices.

Figure 1. Illustration of how `ViewGroup` objects form branches in the layout and contain other `View` objects.

Create a Linear Layout

1. In Android Studio, from the `res/layout` directory, open the `activity_my.xml` file.

   The BlankActivity template you chose when you created this project includes the `activity_my.xml` file with a `RelativeLayout` root view and a `TextView` child view.

2. In the **Preview** pane, click the Hide icon to close the Preview pane.

   In Android Studio, when you open a layout file, you're first shown the Preview pane. Clicking elements in this pane opens the

WYSIWYG tools in the Design pane. For this lesson, you're going to work directly with the XML.

3. Delete the `<TextView>` element.
4. Change the `<RelativeLayout>` element to `<LinearLayout>`.
5. Add the `android:orientation` attribute and set it to `"horizontal"`.
6. Remove the `android:padding` attributes and the `tools:context` attribute.

The result looks like this:

res/layout/activity_my.xml

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal" >
</LinearLayout>
```

`LinearLayout` is a view group (a subclass of `ViewGroup`) that lays out child views in either a vertical or horizontal orientation, as specified by the `android:orientation` attribute. Each child of a `LinearLayout` appears on the screen in the order in which it appears in the XML.

Two other attributes, `android:layout_width` and `android:layout_height`, are required for all views in order to specify their size.

Because the `LinearLayout` is the root view in the layout, it should fill the entire screen area that's available to the app by setting the width and height to `"match_parent"`. This value declares that the view should expand its width or height to *match* the width or height of the parent view.

For more information about layout properties, see the Layout guide.

Add a Text Field

As with every `View` object, you must define certain XML attributes to specify the `EditText` object's properties.

1. In the `activity_my.xml` file, within the `<LinearLayout>` element, define an `<EditText>` element with the `id` attribute set to `@+id/edit_message`.
2. Define the `layout_width` and `layout_height` attributes as `wrap_content`.
3. Define a `hint` attribute as a string object named `edit_message`.

The `<EditText>` element should read as follows:

res/layout/activity_my.xml

```
<EditText android:id="@+id/edit_message"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:hint="@string/edit_message" />
```

Here are the `<EditText>` attributes you added:

`android:id`

This provides a unique identifier for the view, which you can use to reference the object from your app code, such as to read and manipulate the object (you'll see this in the next lesson).

The at sign (`@`) is required when you're referring to any resource object from XML. It is followed by the resource type (`id` in this case), a slash, then the resource name (`edit_message`).

Resource Objects

A resource object is a unique integer name that's associated with an app resource, such as a bitmap, layout file, or string.

Every resource has a corresponding resource object defined in your project's `gen/R.java` file. You can use the object names in the `R` class to refer to your resources, such as when you need to specify a string value for the `android:hint` attribute. You can also create arbitrary resource IDs that you associate with a view using the `android:id` attribute, which allows you to reference that view from other code.

The SDK tools generate the `R.java` file each time you compile your app. You should never modify this file by hand.

For more information, read the guide to Providing Resources.

The plus sign (`+`) before the resource type is needed only when you're defining a resource ID for the first time. When you compile the app, the SDK tools use the ID name to create a new resource ID in your project's `gen/R.java` file that refers to the `EditText` element. With the resource ID declared once this way, other references to the ID do not need the plus sign. Using the plus sign is necessary only when specifying a new resource ID and not needed for concrete resources such as strings or layouts. See the sidebox for more information about resource objects.

`android:layout_width` and `android:layout_height`

Instead of using specific sizes for the width and height, the `"wrap_content"` value specifies that the view should be only as big as needed to fit the contents of the view. If you were to instead use `"match_parent"`, then the `EditText` element would fill the screen, because it would match the size of the parent `LinearLayout`. For more information, see the Layouts guide.

`android:hint`

This is a default string to display when the text field is empty. Instead of using a hard-coded string as the value, the `"@string/edit_message"` value refers to a string resource defined in a separate file. Because this refers to a concrete resource (not just an identifier), it does not need the plus sign. However, because you haven't defined the string resource yet, you'll see a compiler error at first. You'll fix this in the next section by defining the string.

Note: This string resource has the same name as the element ID: `edit_message`. However, references to resources are always scoped by the resource type (such as `id` or `string`), so using the same name does not cause collisions.

Add String Resources

By default, your Android project includes a string resource file at `res/values/strings.xml`. Here, you'll add a new string named `"edit_message"` and set the value to "Enter a message."

1. In Android Studio, from the `res/values` directory, open `strings.xml`.
2. Add a line for a string named `"edit_message"` with the value, "Enter a message".
3. Add a line for a string named `"button_send"` with the value, "Send".

   You'll create the button that uses this string in the next section.

4. Remove the line for the `"hello world"` string.

The result for `strings.xml` looks like this:

res/values/strings.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">My First App</string>
    <string name="edit_message">Enter a message</string>
    <string name="button_send">Send</string>
    <string name="action_settings">Settings</string>
    <string name="title_activity_main">MainActivity</string>
</resources>
```

For text in the user interface, always specify each string as a resource. String resources allow you to manage all UI text in a single location, which makes the text easier to find and update. Externalizing the strings also allows you to localize your app to different languages by providing alternative definitions for each string resource.

For more information about using string resources to localize your app for other languages, see the Supporting Different Devices class.

Add a Button

1. In Android Studio, from the `res/layout` directory, edit the `activity_my.xml` file.
2. Within the `<LinearLayout>` element, define a `<Button>` element immediately following the `<EditText>` element.

3. Set the button's width and height attributes to `"wrap_content"` so the button is only as big as necessary to fit the button's text label.
4. Define the button's text label with the `android:text` attribute; set its value to the `button_send` string resource you defined in the previous section.

Your `<LinearLayout>` should look like this:

res/layout/activity_my.xml

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal" >
      <EditText android:id="@+id/edit_message"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:hint="@string/edit_message" />
      <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/button_send" />
</LinearLayout>
```

Note: This button doesn't need the `android:id` attribute, because it won't be referenced from the activity code.

The layout is currently designed so that both the `EditText` and `Button` widgets are only as big as necessary to fit their content, as shown in figure 2.

Figure 2. The `EditText` and `Button` widgets have their widths set to `"wrap_content"`.

This works fine for the button, but not as well for the text field, because the user might type something longer. It would be nice to fill the unused screen width with the text field. You can do this inside a `LinearLayout` with the *weight* property, which you can specify using the `android:layout_weight` attribute.

The weight value is a number that specifies the amount of remaining space each view should consume, relative to the amount consumed by sibling views. This works kind of like the amount of ingredients in a drink recipe: "2 parts soda, 1 part syrup" means two-thirds of the drink is soda. For example, if you give one view a weight of 2 and another one a weight of 1, the sum is 3, so the first view fills 2/3 of the remaining space and the second view fills the rest. If you add a third view and give it a weight of 1, then the first view (with weight of 2) now gets 1/2 the remaining space, while the remaining two each get 1/4.

The default weight for all views is 0, so if you specify any weight value greater than 0 to only one view, then that view fills whatever space remains after all views are given the space they require.

Make the Input Box Fill in the Screen Width

To fill the remaining space in your layout with the `EditText` element, do the following:

1. In the `activity_my.xml` file, assign the `<EditText>` element's `layout_weight` attribute a value of `1`.
2. Also, assign `<EditText>` element's `layout_width` attribute a value of `0dp`.

   res/layout/activity_my.xml

   ```
   <EditText
       android:layout_weight="1"
       android:layout_width="0dp"
       ... />
   ```

   To improve the layout efficiency when you specify the weight, you should change the width of the `EditText` to be zero (0dp). Setting the width to zero improves layout performance because using `"wrap_content"` as the width requires the system to calculate a width that is ultimately irrelevant because the weight value requires another width calculation to fill the remaining space.

   Figure 3 shows the result when you assign all weight to the `EditText`

element.

Figure 3. The `EditText` widget is given all the layout weight, so it fills the remaining space in the `LinearLayout`.

Here's how your complete `activity_my.xml`layout file should now look:

res/layout/activity_my.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal">
    <EditText android:id="@+id/edit_message"
        android:layout_weight="1"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:hint="@string/edit_message" />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/button_send" />
</LinearLayout>
```

Run Your App

This layout is applied by the default `Activity` class that the SDK tools generated when you created the project. Run the app to see the results:

- In Android Studio, from the toolbar, click **Run** ▶.
- Or from a command line, change directories to the root of your Android project and execute:

  ```
  ant debug
  adb install bin/MyFirstApp-debug.apk
  ```

Continue to the next lesson to learn how to respond to button presses, read content from the text field, start another activity, and more.

Next Get started Next class

Starting Another Activity

Previous Next Get started

This lesson teaches you to

1. Respond to the Send Button
2. Build an Intent
3. Create the Second Activity
4. Receive the Intent
5. Display the Message

You should also read

- Installing the SDK

After completing the previous lesson, you have an app that shows an activity (a single screen) with a text field and a button. In this lesson, you'll add some code to `MyActivity` that starts a new activity when the user clicks the Send button.

Respond to the Send Button

1. In Android Studio, from the `res/layout` directory, edit the `activity_my.xml` file.
2. To the `<Button>` element, add the `android:onClick` attribute.

   res/layout/activity_my.xml

   ```
   <Button
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:text="@string/button_send"
       android:onClick="sendMessage" />
   ```

   The `android:onClick` attribute's value, `"sendMessage"`, is the name of a method in your activity that the system calls when the user clicks the button.

3. In the `java/com.mycompany.myfirstapp` directory, open the `MyActivity.java` file.
4. Within the `MyActivity` class, add the `sendMessage()` method stub shown below.

   java/com.mycompany.myfirstapp/MyActivity.java

   ```
   /** Called when the user clicks the Send button */
   public void sendMessage(View view) {
   ```

```
        // Do something in response to button
    }
```

In order for the system to match this method to the method name given to `android:onClick`, the signature must be exactly as shown. Specifically, the method must:

- Be public
- Have a void return value
- Have a `View` as the only parameter (this will be the `View` that was clicked)

Next, you'll fill in this method to read the contents of the text field and deliver that text to another activity.

Build an Intent

1. In `MyActivity.java`, inside the `sendMessage()` method, create an `Intent` to start an activity called `DisplayMessageActivity` with the following code:

   java/com.mycompany.myfirstapp/MyActivity.java

   ```
   public void sendMessage(View view) {
     Intent intent = new Intent(this, DisplayMessageActivity.class);
   }
   ```

   ### Intents

   An `Intent` is an object that provides runtime binding between separate components (such as two activities). The `Intent` represents an app's "intent to do something." You can use intents for a wide variety of tasks, but most often they're used to start another activity. For more information, see Intents and Intent Filters.

   Note: The reference to `DisplayMessageActivity` will raise an error if you're using an IDE such as Android Studio because the class doesn't exist yet. Ignore the error for now; you'll create the class soon.

   The constructor used here takes two parameters:

   - A `Context` as its first parameter (`this` is used because the `Activity` class is a subclass of `Context`)
   - The `Class` of the app component to which the system should deliver the `Intent` (in this case, the activity that should be started)

   Android Studio indicates that you must import the `Intent` class.

2. At the top of the file, import the `Intent` class:

java/com.mycompany.myfirstapp/MyActivity.java

```
import android.content.Intent;
```

Tip: In Android Studio, press Alt + Enter (option + return on Mac) to import missing classes.

3. Inside the `sendMessage()` method, use `findViewById()` to get the `EditText` element.

   java/com.mycompany.myfirstapp/MyActivity.java

```
public void sendMessage(View view) {
  Intent intent = new Intent(this, DisplayMessageActivity.class);
  EditText editText = (EditText) findViewById(R.id.edit_message);
}
```

4. At the top of the file, import the `EditText` class.

   In Android Studio, press Alt + Enter (option + return on Mac) to import missing classes.

5. Assign the text to a local `message` variable, and use the `putExtra()` method to add its text value to the intent.

   java/com.mycompany.myfirstapp/MyActivity.java

```
public void sendMessage(View view) {
  Intent intent = new Intent(this, DisplayMessageActivity.class);
  EditText editText = (EditText) findViewById(R.id.edit_message);
  String message = editText.getText().toString();
  intent.putExtra(EXTRA_MESSAGE, message);
}
```

   An `Intent` can carry data types as key-value pairs called *extras*. The `putExtra()` method takes the key name in the first parameter and the value in the second parameter.

6. At the top of the `MyActivity` class, add the `EXTRA_MESSAGE` definition as follows:

   java/com.mycompany.myfirstapp/MyActivity.java

```
public class MyActivity extends ActionBarActivity {
    public final static String EXTRA_MESSAGE =
"com.mycompany.myfirstapp.MESSAGE";
    ...
}
```

   For the next activity to query the extra data, you should define the key for your intent's extra using a public constant. It's generally a good practice to define keys for intent extras using your app's package name as a prefix. This ensures the keys are unique, in case your app interacts with other apps.

7. In the `sendMessage()` method, to finish the intent, call the `startActivity()` method, passing it the `Intent` object created in step 1.

With this new code, the complete `sendMessage()` method that's invoked by the Send button now looks like this:

java/com.mycompany.myfirstapp/MyActivity.java

```
/** Called when the user clicks the Send button */
public void sendMessage(View view) {
    Intent intent = new Intent(this, DisplayMessageActivity.class);
    EditText editText = (EditText) findViewById(R.id.edit_message);
    String message = editText.getText().toString();
    intent.putExtra(EXTRA_MESSAGE, message);
    startActivity(intent);
}
```

The system receives this call and starts an instance of the `Activity` specified by the `Intent`. Now you need to create the `DisplayMessageActivity` class in order for this to work.

Create the Second Activity

All subclasses of `Activity` must implement the `onCreate()` method. This method is where the activity receives the intent with the message, then renders the message. Also, the `onCreate()` method must define the activity layout with the `setContentView()` method. This is where the activity performs the initial setup of the activity components.

Create a new activity using Android Studio

Figure 1. The new activity wizard in Android Studio.

Android Studio includes a stub for the `onCreate()` method when you create a new activity.

1. In Android Studio, in the `java` directory, select the package, **com.mycompany.myfirstapp**, right-click, and select **New > Activity > Blank Activity**.
2. In the **Choose options** window, fill in the activity details:
   - **Activity Name**: DisplayMessageActivity
   - **Layout Name**: activity_display_message
   - **Title**: My Message
   - **Hierarchical Parent**: com.mycompany.myfirstapp.MyActivity
   - **Package name**: com.mycompany.myfirstapp

   Click **Finish**.

3. Open the `DisplayMessageActivity.java` file.

   The class already includes an implementation of the required `onCreate()` method. You will update the implementation of this method later. It also includes an

implementation of `onOptionsItemSelected()`, which handles the action bar's *Up* behavior. Keep these two methods as they are for now.

4. Remove the `onCreateOptionsMenu()` method.

   You won't need it for this app.

If you're developing with Android Studio, you can run the app now, but not much happens. Clicking the Send button starts the second activity, but it uses a default "Hello world" layout provided by the template. You'll soon update the activity to instead display a custom text view.

Create the activity without Android Studio

If you're using a different IDE or the command line tools, do the following:

1. Create a new file named `DisplayMessageActivity.java` in the project's `src/` directory, next to the original `MyActivity.java` file.
2. Add the following code to the file:

```
public class DisplayMessageActivity extends ActionBarActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_display_message);

        if (savedInstanceState == null) {
            getSupportFragmentManager().beginTransaction()
                .add(R.id.container, new PlaceholderFragment()).commit();
        }
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle action bar item clicks here. The action bar will
        // automatically handle clicks on the Home/Up button, so long
        // as you specify a parent activity in AndroidManifest.xml.
        int id = item.getItemId();
        if (id == R.id.action_settings) {
            return true;
        }
        return super.onOptionsItemSelected(item);
    }

    /**
     * A placeholder fragment containing a simple view.
     */
    public static class PlaceholderFragment extends Fragment {

        public PlaceholderFragment() { }

        @Override
        public View onCreateView(LayoutInflater inflater, ViewGroup
container,
                Bundle savedInstanceState) {
            View rootView =
inflater.inflate(R.layout.fragment_display_message,
                    container, false);
            return rootView;
```

```
        }
    }
}
```

Note: If you are using an IDE other than Android Studio, your project does not contain the `activity_display_message` layout that's requested by `setContentView()`. That's OK because you will update this method later and won't be using that layout.

3. To your `strings.xml` file, add the new activity's title as follows:

```
<resources>
    ...
    <string name="title_activity_display_message">My Message</string>
</resources>
```

4. In your manifest file, `AndroidManifest.xml`, within the `Application` element, add the `<activity>` element for your `DisplayMessageActivity` class, as follows:

```
<application ... >
    ...
    <activity
        android:name="com.mycompany.myfirstapp.DisplayMessageActivity"
        android:label="@string/title_activity_display_message"
        android:parentActivityName="com.mycompany.myfirstapp.MyActivity" >
        <meta-data
            android:name="android.support.PARENT_ACTIVITY"
            android:value="com.mycompany.myfirstapp.MyActivity" />
    </activity>
</application>
```

The `android:parentActivityName` attribute declares the name of this activity's parent activity within the app's logical hierarchy. The system uses this value to implement default navigation behaviors, such as Up navigation on Android 4.1 (API level 16) and higher. You can provide the same navigation behaviors for older versions of Android by using the Support Library and adding the `<meta-data>` element as shown here.

Note: Your Android SDK should already include the latest Android Support Library, which you installed during the Adding SDK Packages step. When using the templates in Android Studio, the Support Library is automatically added to your app project (you can see the library's JAR file listed under *Android Dependencies*). If you're not using Android Studio, you need to manually add the library to your project—follow the guide for setting up the Support Library then return here.

If you're using a different IDE than Android Studio, don't worry that the app won't yet compile. You'll soon update the activity to display a custom text view.

Receive the Intent

Every `Activity` is invoked by an `Intent`, regardless of how the user navigated there. You can get the `Intent` that started your activity by calling `getIntent()` and retrieve the data contained within the intent.

1. In the `java/com.mycompany.myfirstapp` directory, edit the `DisplayMessageActivity.java` file.
2. In the `onCreate()` method, remove the following line:

   ```
   setContentView(R.layout.activity_display_message);
   ```

3. Get the intent and assign it to a local variable.

   ```
   Intent intent = getIntent();
   ```

4. At the top of the file, import the `Intent` class.

   In Android Studio, press Alt + Enter (option + return on Mac) to import missing classes.

5. Extract the message delivered by `MyActivity` with the `getStringExtra()` method.

   ```
   String message = intent.getStringExtra(MyActivity.EXTRA_MESSAGE);
   ```

Display the Message

1. In the `onCreate()` method, create a `TextView` object.

   ```
   TextView textView = new TextView(this);
   ```

2. Set the text size and message with `setText()`.

   ```
   textView.setTextSize(40);
   textView.setText(message);
   ```

3. Then add the `TextView` as the root view of the activity's layout by passing it to `setContentView()`.

   ```
   setContentView(textView);
   ```

4. At the top of the file, import the `TextView` class.

   In Android Studio, press Alt + Enter (option + return on Mac) to import missing classes.

The complete `onCreate()` method for `DisplayMessageActivity` now looks like this:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    // Get the message from the intent
    Intent intent = getIntent();
    String message = intent.getStringExtra(MyActivity.EXTRA_MESSAGE);

    // Create the text view
    TextView textView = new TextView(this);
    textView.setTextSize(40);
    textView.setText(message);

    // Set the text view as the activity layout
```

```
    setContentView(textView);
}
```

You can now run the app. When it opens, type a message in the text field, click Send, and the message appears on the second activity.

Figure 2. Both activities in the final app, running on Android 4.4.

That's it, you've built your first Android app!

To learn more, follow the link below to the next class.

Next Get started Next class

Adding the Action Bar

Previous Next Get started

Dependencies and prerequisites

- Android 2.1 or higher

You should also read

- Action Bar
- Implementing Effective Navigation

Design Guide

Action Bar

The action bar is one of the most important design elements you can implement for your app's activities. It provides several user interface features that make your app immediately familiar to users by offering consistency between other Android apps. Key functions include:

- A dedicated space for giving your app an identity and indicating the user's location in the app.
- Access to important actions in a predictable way (such as Search).
- Support for navigation and view switching (with tabs or drop-down lists).

This training class offers a quick guide to the action bar's basics. For more information about action bar's various features, see the Action Bar guide.

Lessons

Setting Up the Action Bar

Learn how to add a basic action bar to your activity, whether your app supports only Android 3.0 and higher or also supports versions as low as Android 2.1 (by using the Android Support Library).

Adding Action Buttons

Learn how to add and respond to user actions in the action bar.

Styling the Action Bar

Learn how to customize the appearance of your action bar.

Overlaying the Action Bar

Learn how to overlay the action bar in front of your layout, allowing for seamless transitions when hiding the action bar.

Next Get started Next class

Setting Up the Action Bar

Previous Next Get started

This lesson teaches you to

1. Support Android 3.0 and Above Only
2. Support Android 2.1 and Above

You should also read

- Setting Up the Support Library

In its most basic form, the action bar displays the title for the activity and the app icon on the left. Even in this simple form, the action bar is useful for all activities to inform users about where they are and to maintain a consistent identity for your app.

Figure 1. An action bar with the app icon and activity title.

Setting up a basic action bar requires that your app use an activity theme that enables the action bar. How to request such a theme depends on which version of Android is the lowest supported by your app. So this lesson is divided into two sections depending on which Android version is your lowest supported.

Support Android 3.0 and Above Only

Beginning with Android 3.0 (API level 11), the action bar is included in all activities that use the `Theme.Holo` theme (or one of its descendants), which is the default theme when either the `targetSdkVersion` or `minSdkVersion` attribute is set to `"11"` or greater.

So to add the action bar to your activities, simply set either attribute to `11` or higher. For example:

```
<manifest ... >
    <uses-sdk android:minSdkVersion="11" ... />
    ...
</manifest>
```

Note: If you've created a custom theme, be sure it uses one of the `Theme.Holo` themes as its parent. For details, see Styling the Action Bar.

Now the `Theme.Holo` theme is applied to your app and all activities show the action bar. That's it.

Support Android 2.1 and Above

Adding the action bar when running on versions older than Android 3.0 (down to Android 2.1) requires that you include the Android Support Library in your application.

To get started, read the Support Library Setup document and set up the **v7 appcompat** library (once you've downloaded the library package, follow the instructions for Adding libraries with resources).

Once you have the Support Library integrated with your app project:

1. Update your activity so that it extends `ActionBarActivity`. For example:

   ```
   public class MainActivity extends ActionBarActivity { ... }
   ```

2. In your manifest file, update either the `<application>` element or individual `<activity>` elements to use one of the `Theme.AppCompat` themes. For example:

   ```
   <activity android:theme="@style/Theme.AppCompat.Light" ... >
   ```

> **Note:** If you've created a custom theme, be sure it uses one of the `Theme.AppCompat` themes as its parent. For details, see Styling the Action Bar.

Now your activity includes the action bar when running on Android 2.1 (API level 7) or higher.

Remember to properly set your app's API level support in the manifest:

```
<manifest ... >
    <uses-sdk android:minSdkVersion="7"
android:targetSdkVersion="18" />
    ...
</manifest>
```

Next Get started Next class

Adding Action Buttons

Previous Next Get started

This lesson teaches you to

1. Specify the Actions in XML
2. Add the Actions to the Action Bar
3. Respond to Action Buttons
4. Add Up Button for Low-level Activities

You should also read

- Providing Up Navigation

The action bar allows you to add buttons for the most important action items relating to the app's current context. Those that appear directly in the action bar with an icon and/or text are known as *action buttons*. Actions that can't fit in the action bar or aren't important enough are hidden in the action overflow.

Figure 1. An action bar with an action button for Search and the action overflow, which reveals additional actions.

Specify the Actions in XML

All action buttons and other items available in the action overflow are

defined in an XML menu resource. To add actions to the action bar, create a new XML file in your project's `res/menu/` directory.

Add an `<item>` element for each item you want to include in the action bar. For example:

res/menu/main_activity_actions.xml

```
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
    <!-- Search, should appear as action button -->
    <item android:id="@+id/action_search"
          android:icon="@drawable/ic_action_search"
          android:title="@string/action_search"
          android:showAsAction="ifRoom" />
    <!-- Settings, should always be in the overflow -->
    <item android:id="@+id/action_settings"
          android:title="@string/action_settings"
          android:showAsAction="never" />
</menu>
```

Download action bar icons

To best match the Android iconography guidelines, you should use icons provided in the Action Bar Icon Pack.

This declares that the Search action should appear as an action button when room is available in the action bar, but the Settings action should always appear in the overflow. (By default, all actions appear in the overflow, but it's good practice to explicitly declare your design intentions for each action.)

The `icon` attribute requires a resource ID for an image. The name that follows `@drawable/` must be the name of a bitmap image you've saved in your project's `res/drawable/` directory. For example, `"@drawable/ic_action_search"` refers to `ic_action_search.png`. Likewise, the `title` attribute uses a string resource that's defined by an XML file in your project's `res/values/` directory, as discussed in Building a Simple User Interface.

Note: When creating icons and other bitmap images for your app, it's important that you provide multiple versions that are each optimized for a different screen density. This is discussed more in the lesson about

Supporting Different Screens.

If your app is using the Support Library for compatibility on versions as low as Android 2.1, the `showAsAction` attribute is not available from the `android:` namespace. Instead this attribute is provided by the Support Library and you must define your own XML namespace and use that namespace as the attribute prefix. (A custom XML namespace should be based on your app name, but it can be any name you want and is only accessible within the scope of the file in which you declare it.) For example:

res/menu/main_activity_actions.xml

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:yourapp="http://schemas.android.com/apk/res-auto" >
    <!-- Search, should appear as action button -->
    <item android:id="@+id/action_search"
          android:icon="@drawable/ic_action_search"
          android:title="@string/action_search"
          yourapp:showAsAction="ifRoom"  />
    ...
</menu>
```

Add the Actions to the Action Bar

To place the menu items into the action bar, implement the `onCreateOptionsMenu()` callback method in your activity to inflate the menu resource into the given `Menu` object. For example:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu items for use in the action bar
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.main_activity_actions, menu);
    return super.onCreateOptionsMenu(menu);
}
```

Respond to Action Buttons

When the user presses one of the action buttons or another item in the action overflow, the system calls your activity's `onOptionsItemSelected()` callback method. In your implementation of this method, call `getItemId()` on the given `MenuItem` to determine which item was pressed—the

returned ID matches the value you declared in the corresponding `<item>` element's `android:id` attribute.

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Handle presses on the action bar items
    switch (item.getItemId()) {
        case R.id.action_search:
            openSearch();
            return true;
        case R.id.action_settings:
            openSettings();
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

Add Up Button for Low-level Activities

Figure 4. The *Up* button in Gmail.

All screens in your app that are not the main entrance to your app (activities that are not the "home" screen) should offer the user a way to navigate to the logical parent screen in the app's hierarchy by pressing the *Up* button in the action bar.

When running on Android 4.1 (API level 16) or higher, or when using `ActionBarActivity` from the Support Library, performing *Up* navigation simply requires that you declare the parent activity in the manifest file and enable the *Up* button for the action bar.

For example, here's how you can declare an activity's parent in the manifest:

```
<application ... >
    ...
    <!-- The main/home activity (it has no parent activity) -->
```

```
    <activity
        android:name="com.example.myfirstapp.MainActivity" ...>
        ...
    </activity>
    <!-- A child of the main activity -->
    <activity
        android:name="com.example.myfirstapp.DisplayMessageActivity"
        android:label="@string/title_activity_display_message"

android:parentActivityName="com.example.myfirstapp.MainActivity" >
        <!-- Parent activity meta-data to support 4.0 and lower -->
        <meta-data
            android:name="android.support.PARENT_ACTIVITY"
            android:value="com.example.myfirstapp.MainActivity" />
    </activity>
</application>
```

Then enable the app icon as the *Up* button by calling `setDisplayHomeAsUpEnabled()`:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_displaymessage);

    getSupportActionBar().setDisplayHomeAsUpEnabled(true);
    // If your minSdkVersion is 11 or higher, instead use:
    // getActionBar().setDisplayHomeAsUpEnabled(true);
}
```

Because the system now knows `MainActivity` is the parent activity for `DisplayMessageActivity`, when the user presses the *Up* button, the system navigates to the parent activity as appropriate—you **do not** need to handle the *Up* button's event.

For more information about up navigation, see Providing Up Navigation.

Next Get started Next class

Styling the Action Bar

Previous Next Get started

This lesson teaches you to

1. Use an Android Theme
2. Customize the Background
3. Customize the Text Color
4. Customize the Tab Indicator

You should also read

- Styles and Themes
- Android Action Bar Style Generator

The action bar provides your users a familiar and predictable way to perform actions and navigate your app, but that doesn't mean it needs to look exactly the same as it does in other apps. If you want to style the action bar to better fit your product brand, you can easily do so using Android's style and theme resources.

Android includes a few built-in activity themes that include "dark" or "light" action bar styles. You can also extend these themes to further customize the look for your action bar.

Note: If you are using the Support Library APIs for the action bar, then you must use (or override) the `Theme.AppCompat` family of styles (rather than the `Theme.Holo` family, available in API level 11 and higher). In doing so, each style property that you declare must be declared twice: once using the platform's style properties (the `android:` properties) and once using the style properties included in the Support Library (the `appcompat.R.attr` properties—the context for these properties is actually

your app). See the examples below for details.

Use an Android Theme

Android includes two baseline activity themes that dictate the color for the action bar:

- `Theme.Holo` for a "dark" theme.
- `Theme.Holo.Light` for a "light" theme.

You can apply these themes to your entire app or to individual activities by declaring them in your manifest file with the `android:theme` attribute for the `<application>` element or individual `<activity>` elements.

For example:

```
<application android:theme="@android:style/Theme.Holo.Light" ... />
```

You can also use a dark action bar while the rest of the activity uses the light color scheme by declaring the `Theme.Holo.Light.DarkActionBar` theme.

When using the Support Library, you must instead use the `Theme.AppCompat` themes:

- `Theme.AppCompat` for the "dark" theme.
- `Theme.AppCompat.Light` for the "light" theme.
- `Theme.AppCompat.Light.DarkActionBar` for the light theme with a dark action bar.

Be sure that you use action bar icons that properly contrast with the color of your action bar. To help you, the Action Bar Icon Pack includes standard action icons for use with both the Holo light and Holo dark action bar.

Customize the Background

To change the action bar background, create a custom theme for your activity that overrides the `actionBarStyle` property. This property points to another style in which you can override the `background` property to specify a drawable resource for the action bar background.

If your app uses navigation tabs or the split action bar, then you can also specify the background for these bars using the `backgroundStacked` and `backgroundSplit` properties, respectively.

Caution: It's important that you declare an appropriate parent theme from which your custom theme and style inherit their styles. Without a parent style, your action bar will be without many style properties unless you explicitly declare them yourself.

For Android 3.0 and higher only

When supporting Android 3.0 and higher only, you can define the action bar's background like this:

res/values/themes.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- the theme applied to the application or activity -->
    <style name="CustomActionBarTheme"
           parent="@android:style/Theme.Holo.Light.DarkActionBar">
        <item name="android:actionBarStyle">@style/MyActionBar</item>
    </style>

    <!-- ActionBar styles -->
    <style name="MyActionBar"

parent="@android:style/Widget.Holo.Light.ActionBar.Solid.Inverse">
        <item
name="android:background">@drawable/actionbar_background</item>
    </style>
</resources>
```

Then apply your theme to your entire app or individual activities:

```
<application android:theme="@style/CustomActionBarTheme" ... />
```

For Android 2.1 and higher

When using the Support Library, the same theme as above must instead look like this:

res/values/themes.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- the theme applied to the application or activity -->
    <style name="CustomActionBarTheme"
           parent="@style/Theme.AppCompat.Light.DarkActionBar">
        <item name="android:actionBarStyle">@style/MyActionBar</item>

        <!-- Support library compatibility -->
        <item name="actionBarStyle">@style/MyActionBar</item>
    </style>

    <!-- ActionBar styles -->
    <style name="MyActionBar"

parent="@style/Widget.AppCompat.Light.ActionBar.Solid.Inverse">
        <item
name="android:background">@drawable/actionbar_background</item>

        <!-- Support library compatibility -->
        <item name="background">@drawable/actionbar_background</item>
    </style>
</resources>
```

Then apply your theme to your entire app or individual activities:

```
<application android:theme="@style/CustomActionBarTheme" ... />
```

Customize the Text Color

To modify the color of text in the action bar, you need to override separate properties for each text element:

- Action bar title: Create a custom style that specifies the `textColor` property and specify that style for the `titleTextStyle` property in your custom `actionBarStyle`.

  Note: The custom style applied to `titleTextStyle` should use `TextAppearance.Holo.Widget.ActionBar.Title` as the parent style.

- Action bar tabs: Override `actionBarTabTextStyle` in your activity theme.
- Action buttons: Override `actionMenuTextColor` in your activity theme.

For Android 3.0 and higher only

When supporting Android 3.0 and higher only, your style XML file might look like this:

res/values/themes.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- the theme applied to the application or activity -->
    <style name="CustomActionBarTheme"
           parent="@style/Theme.Holo">
        <item name="android:actionBarStyle">@style/MyActionBar</item>
        <item
name="android:actionBarTabTextStyle">@style/MyActionBarTabText</item>
        <item
name="android:actionMenuTextColor">@color/actionbar_text</item>
    </style>

    <!-- ActionBar styles -->
    <style name="MyActionBar"
           parent="@style/Widget.Holo.ActionBar">
        <item
name="android:titleTextStyle">@style/MyActionBarTitleText</item>
    </style>

    <!-- ActionBar title text -->
    <style name="MyActionBarTitleText"
```

```
parent="@style/TextAppearance.Holo.Widget.ActionBar.Title">
        <item name="android:textColor">@color/actionbar_text</item>
    </style>

    <!-- ActionBar tabs text styles -->
    <style name="MyActionBarTabText"
           parent="@style/Widget.Holo.ActionBar.TabText">
        <item name="android:textColor">@color/actionbar_text</item>
    </style>
</resources>
```

For Android 2.1 and higher

When using the Support Library, your style XML file might look like this:

res/values/themes.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- the theme applied to the application or activity -->
    <style name="CustomActionBarTheme"
           parent="@style/Theme.AppCompat">
        <item name="android:actionBarStyle">@style/MyActionBar</item>
        <item
name="android:actionBarTabTextStyle">@style/MyActionBarTabText</item>
        <item
name="android:actionMenuTextColor">@color/actionbar_text</item>

        <!-- Support library compatibility -->
        <item name="actionBarStyle">@style/MyActionBar</item>
        <item
name="actionBarTabTextStyle">@style/MyActionBarTabText</item>
        <item name="actionMenuTextColor">@color/actionbar_text</item>
    </style>

    <!-- ActionBar styles -->
    <style name="MyActionBar"
           parent="@style/Widget.AppCompat.ActionBar">
        <item
name="android:titleTextStyle">@style/MyActionBarTitleText</item>

        <!-- Support library compatibility -->
        <item
name="titleTextStyle">@style/MyActionBarTitleText</item>
    </style>

    <!-- ActionBar title text -->
    <style name="MyActionBarTitleText"

parent="@style/TextAppearance.AppCompat.Widget.ActionBar.Title">
        <item name="android:textColor">@color/actionbar_text</item>
        <!-- The textColor property is backward compatible with the
Support Library -->
    </style>

    <!-- ActionBar tabs text -->
    <style name="MyActionBarTabText"
```

```
            parent="@style/Widget.AppCompat.ActionBar.TabText">
        <item name="android:textColor">@color/actionbar_text</item>
        <!-- The textColor property is backward compatible with the
Support Library -->
    </style>
</resources>
```

Customize the Tab Indicator

To change the indicator used for the navigation tabs, create an activity theme that overrides the `actionBarTabStyle` property. This property points to another style resource in which you override the `background` property that should specify a state-list drawable.

Note: A state-list drawable is important so that the tab currently selected indicates its state with a background different than the other tabs. For more information about how to create a drawable resource that handles multiple button states, read the State List documentation.

For example, here's a state-list drawable that declares a specific background image for several different states of an action bar tab:

res/drawable/actionbar_tab_indicator.xml

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">

<!-- STATES WHEN BUTTON IS NOT PRESSED -->

    <!-- Non focused states -->
    <item android:state_focused="false"
android:state_selected="false"
          android:state_pressed="false"
          android:drawable="@drawable/tab_unselected" />
    <item android:state_focused="false" android:state_selected="true"
          android:state_pressed="false"
          android:drawable="@drawable/tab_selected" />

    <!-- Focused states (such as when focused with a d-pad or mouse
hover) -->
    <item android:state_focused="true" android:state_selected="false"
          android:state_pressed="false"
```

```
          android:drawable="@drawable/tab_unselected_focused" />
    <item android:state_focused="true" android:state_selected="true"
          android:state_pressed="false"
          android:drawable="@drawable/tab_selected_focused" />


<!-- STATES WHEN BUTTON IS PRESSED -->

    <!-- Non focused states -->
    <item android:state_focused="false"
android:state_selected="false"
          android:state_pressed="true"
          android:drawable="@drawable/tab_unselected_pressed" />
    <item android:state_focused="false" android:state_selected="true"
        android:state_pressed="true"
        android:drawable="@drawable/tab_selected_pressed" />

    <!-- Focused states (such as when focused with a d-pad or mouse
hover) -->
    <item android:state_focused="true" android:state_selected="false"
          android:state_pressed="true"
          android:drawable="@drawable/tab_unselected_pressed" />
    <item android:state_focused="true" android:state_selected="true"
          android:state_pressed="true"
          android:drawable="@drawable/tab_selected_pressed" />
</selector>
```

For Android 3.0 and higher only

When supporting Android 3.0 and higher only, your style XML file might look like this:

res/values/themes.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- the theme applied to the application or activity -->
    <style name="CustomActionBarTheme"
           parent="@style/Theme.Holo">
        <item
name="android:actionBarTabStyle">@style/MyActionBarTabs</item>
    </style>

    <!-- ActionBar tabs styles -->
    <style name="MyActionBarTabs"
           parent="@style/Widget.Holo.ActionBar.TabView">
        <!-- tab indicator -->
        <item
name="android:background">@drawable/actionbar_tab_indicator</item>
    </style>
</resources>
```

For Android 2.1 and higher

When using the Support Library, your style XML file might look like this:

res/values/themes.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- the theme applied to the application or activity -->
    <style name="CustomActionBarTheme"
           parent="@style/Theme.AppCompat">
        <item
name="android:actionBarTabStyle">@style/MyActionBarTabs</item>

        <!-- Support library compatibility -->
        <item name="actionBarTabStyle">@style/MyActionBarTabs</item>
    </style>

    <!-- ActionBar tabs styles -->
    <style name="MyActionBarTabs"
           parent="@style/Widget.AppCompat.ActionBar.TabView">
        <!-- tab indicator -->
        <item
name="android:background">@drawable/actionbar_tab_indicator</item>

        <!-- Support library compatibility -->
        <item
name="background">@drawable/actionbar_tab_indicator</item>
    </style>
</resources>
```

More resources

- See more style properties for the action bar are listed in the Action Bar guide.
- Learn more about how themes work in the Styles and Themes guide.
- For even more complete styling for the action bar, try the Android Action Bar Style Generator.

Next Get started Next class

Overlaying the Action Bar

Previous Next Get started

This lesson teaches you to

1. Enable Overlay Mode
    1. For Android 3.0 and higher only
    2. For Android 2.1 and higher
2. Specify Layout Top-margin

You should also read

- Styles and Themes

By default, the action bar appears at the top of your activity window, slightly reducing the amount of space available for the rest of your activity's layout. If, during the course of user interaction, you want to hide and show the action bar, you can do so by calling `hide()` and `show()` on the `ActionBar`. However, this causes your activity to recompute and redraw the layout based on its new size.

Figure 1. Gallery's action bar in overlay mode.

To avoid resizing your layout when the action bar hides and shows, you can enable *overlay mode* for the action bar. When in overlay mode, your activity layout uses all the space available as if the action bar is not there and the system draws the action bar in front of your layout. This obscures some of the layout at the top, but now when the action bar hides or appears, the system does not need to resize your layout and the transition is seamless.

Tip: If you want your layout to be partially visible behind the action bar, create a custom style for the action bar with a partially transparent background, such as the one shown in figure 1. For information about how to define the action bar background, read Styling the Action Bar.

Enable Overlay Mode

To enable overlay mode for the action bar, you need to create a custom theme that extends an existing action bar theme and set the `android:windowActionBarOverlay` property to `true`.

For Android 3.0 and higher only

If your `minSdkVersion` is set to `11` or higher, your custom theme should use `Theme.Holo` theme (or one of its descendants) as your parent theme. For example:

```
<resources>
    <!-- the theme applied to the application or activity -->
    <style name="CustomActionBarTheme"
           parent="@android:style/Theme.Holo">
        <item name="android:windowActionBarOverlay">true</item>
    </style>
</resources>
```

For Android 2.1 and higher

If your app is using the Support Library for compatibility on devices running versions lower than Android 3.0, your custom theme should use

`Theme.AppCompat` theme (or one of its descendants) as your parent theme. For example:

```
<resources>
    <!-- the theme applied to the application or activity -->
    <style name="CustomActionBarTheme"
           parent="@android:style/Theme.AppCompat">
        <item name="android:windowActionBarOverlay">true</item>

        <!-- Support library compatibility -->
        <item name="windowActionBarOverlay">true</item>
    </style>
</resources>
```

Also notice that this theme includes two definitions for the `windowActionBarOverlay` style: one with the `android:` prefix and one without. The one with the `android:` prefix is for versions of Android that include the style in the platform and the one without the prefix is for older versions that read the style from the Support Library.

Specify Layout Top-margin

When the action bar is in overlay mode, it might obscure some of your layout that should remain visible. To ensure that such items remain below the action bar at all times, add either margin or padding to the top of the view(s) using the height specified by `actionBarSize`. For example:

```
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingTop="?android:attr/actionBarSize">
    ...
</RelativeLayout>
```

If you're using the Support Library for the action bar, you need to remove the `android:` prefix. For example:

```
<!-- Support library compatibility -->
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingTop="?attr/actionBarSize">
    ...
</RelativeLayout>
```

In this case, the `?attr/actionBarSize` value without the prefix works on

all versions, including Android 3.0 and higher.

Next Get started Next class

Supporting Different Devices

Previous Next Get started

Dependencies and prerequisites

- Android 1.6 or higher

You should also read

- Application Resources
- Designing for Multiple Screens

Android devices come in many shapes and sizes all around the world. With a wide range of device types, you have an opportunity to reach a huge audience with your app. In order to be as successful as possible on Android, your app needs to adapt to various device configurations. Some of the important variations that you should consider include different languages, screen sizes, and versions of the Android platform.

This class teaches you how to use basic platform features that leverage alternative resources and other features so your app can provide an optimized user experience on a variety of Android-compatible devices, using a single application package (APK).

Lessons

Supporting Different Languages
: Learn how to support multiple languages with alternative string resources.

Supporting Different Screens
: Learn how to optimize the user experience for different screen sizes and densities.

Supporting Different Platform Versions

Learn how to use APIs available in new versions of Android while continuing to support older versions of Android.

Next Get started Next class

Supporting Different Languages

Previous Next Get started

This class teaches you to

1. Create Locale Directories and String Files
2. Use the String Resources

You should also read

- Localization Checklist
- Localization with Resources

It's always a good practice to extract UI strings from your app code and keep them in an external file. Android makes this easy with a resources directory in each Android project.

If you created your project using the Android SDK Tools (read Creating an Android Project), the tools create a `res/` directory in the top level of the project. Within this `res/` directory are subdirectories for various resource types. There are also a few default files such as `res/values/strings.xml`, which holds your string values.

Create Locale Directories and String Files

To add support for more languages, create additional `values` directories inside `res/` that include a hyphen and the ISO language code at the end of the directory name. For example, `values-es/` is the directory containing simple resourcess for the Locales with the language code "es". Android loads the appropriate resources according to the locale settings of the device at run time. For more information, see Providing Alternative Resources.

Once you've decided on the languages you will support, create the resource subdirectories and string resource files. For example:

```
MyProject/
    res/
       values/
           strings.xml
       values-es/
           strings.xml
       values-fr/
           strings.xml
```

Add the string values for each locale into the appropriate file.

At runtime, the Android system uses the appropriate set of string resources based on the locale currently set for the user's device.

For example, the following are some different string resource files for different languages.

English (default locale), `/values/strings.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="title">My Application</string>
    <string name="hello_world">Hello World!</string>
</resources>
```

Spanish, `/values-es/strings.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="title">Mi Aplicación</string>
    <string name="hello_world">Hola Mundo!</string>
</resources>
```

French, `/values-fr/strings.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="title">Mon Application</string>
    <string name="hello_world">Bonjour le monde !</string>
</resources>
```

Note: You can use the locale qualifier (or any configuration qualifer) on any resource type, such as if you want to provide localized versions of your bitmap drawable. For more information, see Localization.

Use the String Resources

You can reference your string resources in your source code and other XML files using the resource name defined by the `<string>` element's `name` attribute.

In your source code, you can refer to a string resource with the syntax `R.string.<string_name>`. There are a variety of methods that accept a string resource this way.

For example:

```
// Get a string resource from your app's Resources
String hello = getResources().getString(R.string.hello_world);

// Or supply a string resource to a method that requires a string
TextView textView = new TextView(this);
textView.setText(R.string.hello_world);
```

In other XML files, you can refer to a string resource with the syntax `@string/<string_name>` whenever the XML attribute accepts a string value.

For example:

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/hello_world" />
```

Next Get started Next class

Supporting Different Screens

Previous Next Get started

This lesson teaches you to

1. Create Different Layouts
2. Create Different Bitmaps

You should also read

- Designing for Multiple Screens
- Providing Resources
- Iconography design guide

Android categorizes device screens using two general properties: size and density. You should expect that your app will be installed on devices with screens that range in both size and density. As such, you should include some alternative resources that optimize your app's appearance for different screen sizes and densities.

- There are four generalized sizes: small, normal, large, xlarge
- And four generalized densities: low (ldpi), medium (mdpi), high (hdpi), extra high (xhdpi)

To declare different layouts and bitmaps you'd like to use for different screens, you must place these alternative resources in separate directories, similar to how you do for different language strings.

Also be aware that the screens orientation (landscape or portrait) is considered a variation of screen size, so many apps should revise the layout to optimize the user experience in each orientation.

Create Different Layouts

To optimize your user experience on different screen sizes, you should create a unique layout XML file for each screen size you want to support. Each layout should be saved into the appropriate resources directory, named with a `-<screen_size>` suffix. For example, a unique layout for large screens should be saved under `res/layout-large/`.

Note: Android automatically scales your layout in order to properly fit the screen. Thus, your layouts for different screen sizes don't need to worry about the absolute size of UI elements but instead focus on the layout structure that affects the user experience (such as the size or position of important views relative to sibling views).

For example, this project includes a default layout and an alternative layout for *large* screens:

```
MyProject/
    res/
        layout/
            main.xml
        layout-large/
            main.xml
```

The file names must be exactly the same, but their contents are different in order to provide an optimized UI for the corresponding screen size.

Simply reference the layout file in your app as usual:

```
@Override
 protected void onCreate(Bundle savedInstanceState) {
     super.onCreate(savedInstanceState);
     setContentView(R.layout.main);
}
```

The system loads the layout file from the appropriate layout directory based on screen size of the device on which your app is running. More information about how Android selects the appropriate resource is available in the Providing Resources guide.

As another example, here's a project with an alternative layout for landscape orientation:

```
MyProject/
    res/
        layout/
            main.xml
        layout-land/
            main.xml
```

By default, the `layout/main.xml` file is used for portrait orientation.

If you want to provide a special layout for landscape, including while on large screens, then you need to use both the `large` and `land` qualifier:

```
MyProject/
    res/
        layout/                # default (portrait)
            main.xml
        layout-land/           # landscape
            main.xml
        layout-large/          # large (portrait)
            main.xml
        layout-large-land/     # large landscape
            main.xml
```

Note: Android 3.2 and above supports an advanced method of defining screen sizes that allows you to specify resources for screen sizes based on the minimum width and height in terms of density-independent pixels. This lesson does not cover this new technique. For more information, read Designing for Multiple Screens.

Create Different Bitmaps

You should always provide bitmap resources that are properly scaled to each of the generalized density buckets: low, medium, high and extra-high density. This helps you achieve good graphical quality and performance on all screen densities.

To generate these images, you should start with your raw resource in vector format and generate the images for each density using the following size scale:

- xhdpi: 2.0
- hdpi: 1.5
- mdpi: 1.0 (baseline)
- ldpi: 0.75

This means that if you generate a 200x200 image for xhdpi devices, you should generate the same resource in 150x150 for hdpi, 100x100 for mdpi, and 75x75 for ldpi devices.

Then, place the files in the appropriate drawable resource directory:

```
MyProject/
    res/
        drawable-xhdpi/
            awesomeimage.png
        drawable-hdpi/
            awesomeimage.png
        drawable-mdpi/
            awesomeimage.png
        drawable-ldpi/
            awesomeimage.png
```

Any time you reference `@drawable/awesomeimage`, the system selects the appropriate bitmap based on the screen's density.

Note: Low-density (ldpi) resources aren't always necessary. When you provide hdpi assets, the system scales them down by one half to properly fit ldpi screens.

For more tips and guidelines about creating icon assets for your app, see the Iconography design guide.

Next Get started Next class

Supporting Different Platform Versions

Previous Next Get started

This lesson teaches you to

1. Specify Minimum and Target API Levels
2. Check System Version at Runtime
3. Use Platform Styles and Themes

You should also read

- Android API Levels
- Android Support Library

While the latest versions of Android often provide great APIs for your app, you should continue to support older versions of Android until more devices get updated. This lesson shows you how to take advantage of the latest APIs while continuing to support older versions as well.

The dashboard for Platform Versions is updated regularly to show the distribution of active devices running each version of Android, based on the number of devices that visit the Google Play Store. Generally, it's a good practice to support about 90% of the active devices, while targeting your app to the latest version.

Tip: In order to provide the best features and functionality across several Android versions, you should use the Android Support Library in your app, which allows you to use several recent platform APIs on older versions.

Specify Minimum and Target API Levels

The AndroidManifest.xml file describes details about your app and identifies which versions of Android it supports. Specifically, the `minSdkVersion` and `targetSdkVersion` attributes for the `<uses-sdk` element identify the lowest API level with which your app is compatible and the highest API level against which you've designed and tested your app.

For example:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
... >
    <uses-sdk android:minSdkVersion="4" android:targetSdkVersion="15"
/>
    ...
</manifest>
```

As new versions of Android are released, some style and behaviors may change. To allow your app to take advantage of these changes and ensure that your app fits the style of each user's device, you should set the `targetSdkVersion` value to match the latest Android version available.

Check System Version at Runtime

Android provides a unique code for each platform version in the `Build` constants class. Use these codes within your app to build conditions that ensure the code that depends on higher API levels is executed only when those APIs are available on the system.

```
private void setUpActionBar() {
    // Make sure we're running on Honeycomb or higher to use
ActionBar APIs
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
        ActionBar actionBar = getActionBar();
        actionBar.setDisplayHomeAsUpEnabled(true);
    }
}
```

Note: When parsing XML resources, Android ignores XML attributes that aren't supported by the current device. So you can safely use XML attributes that are only supported by newer versions without worrying about older versions breaking when they encounter that code. For example, if you set the `targetSdkVersion="11"`, your app includes the

`ActionBar` by default on Android 3.0 and higher. To then add menu items to the action bar, you need to set `android:showAsAction="ifRoom"` in your menu resource XML. It's safe to do this in a cross-version XML file, because the older versions of Android simply ignore the `showAsAction` attribute (that is, you *do not* need a separate version in `res/menu-v11/`).

Use Platform Styles and Themes

Android provides user experience themes that give apps the look and feel of the underlying operating system. These themes can be applied to your app within the manifest file. By using these built in styles and themes, your app will naturally follow the latest look and feel of Android with each new release.

To make your activity look like a dialog box:

```
<activity android:theme="@android:style/Theme.Dialog">
```

To make your activity have a transparent background:

```
<activity android:theme="@android:style/Theme.Translucent">
```

To apply your own custom theme defined in `/res/values/styles.xml`:

```
<activity android:theme="@style/CustomTheme">
```

To apply a theme to your entire app (all activities), add the `android:theme` attribute to the `<application>` element:

```
<application android:theme="@style/CustomTheme">
```

For more about creating and using themes, read the Styles and Themes guide.

Next Get started Next class

Managing the Activity Lifecycle

Previous Next Get started

Dependencies and prerequisites

- How to create an Android project (see Creating an Android Project)

You should also read

- Activities

Try it out

Download the demo

ActivityLifecycle.zip

As a user navigates through, out of, and back to your app, the `Activity` instances in your app transition between different states in their lifecycle. For instance, when your activity starts for the first time, it comes to the foreground of the system and receives user focus. During this process, the Android system calls a series of lifecycle methods on the activity in which you set up the user interface and other components. If the user performs an action that starts another activity or switches to another app, the system calls another set of lifecycle methods on your activity as it moves into the background (where the activity is no longer visible, but the instance and its state remains intact).

Within the lifecycle callback methods, you can declare how your activity behaves when the user leaves and re-enters the activity. For example, if you're building a streaming video player, you might pause the video and terminate the network connection when the user switches to another

app. When the user returns, you can reconnect to the network and allow the user to resume the video from the same spot.

This class explains important lifecycle callback methods that each `Activity` instance receives and how you can use them so your activity does what the user expects and does not consume system resources when your activity doesn't need them.

Lessons

Starting an Activity
: Learn the basics about the activity lifecycle, how the user can launch your app, and how to perform basic activity creation.

Pausing and Resuming an Activity
: Learn what happens when your activity is paused (partially obscured) and resumed and what you should do during these state changes.

Stopping and Restarting an Activity
: Learn what happens when the user completely leaves your activity and returns to it.

Recreating an Activity
: Learn what happens when your activity is destroyed and how you can rebuild the activity state when necessary.

Next Get started Next class

Starting an Activity

Previous Next Get started

This lesson teaches you to

1. Understand the Lifecycle Callbacks
2. Specify Your App's Launcher Activity
3. Create a New Instance
4. Destroy the Activity

You should also read

- Activities

Try it out

Download the demo

ActivityLifecycle.zip

Unlike other programming paradigms in which apps are launched with a `main()` method, the Android system initiates code in an `Activity` instance by invoking specific callback methods that correspond to specific stages of its lifecycle. There is a sequence of callback methods that start up an activity and a sequence of callback methods that tear down an activity.

This lesson provides an overview of the most important lifecycle methods and shows you how to handle the first lifecycle callback that creates a new instance of your activity.

Understand the Lifecycle Callbacks

During the life of an activity, the system calls a core set of lifecycle methods in a sequence similar to a step pyramid. That is, each stage of the activity lifecycle is a separate step on the pyramid. As the system creates a new activity instance,

each callback method moves the activity state one step toward the top. The top of the pyramid is the point at which the activity is running in the foreground and the user can interact with it.

As the user begins to leave the activity, the system calls other methods that move the activity state back down the pyramid in order to dismantle the activity. In some cases, the activity will move only part way down the pyramid and wait (such as when the user switches to another app), from which point the activity can move back to the top (if the user returns to the activity) and resume where the user left off.

Figure 1. A simplified illustration of the Activity lifecycle, expressed as a step pyramid. This shows how, for every callback used to take the activity a step toward the Resumed state at the top, there's a callback method that takes the activity a step down. The activity can also return to the resumed state from the Paused and Stopped state.

Depending on the complexity of your activity, you probably don't need to implement all the lifecycle methods. However, it's important that you understand each one and implement those that ensure your app behaves the way users expect. Implementing your activity lifecycle methods properly ensures your app behaves well in several ways, including that it:

- Does not crash if the user receives a phone call or switches to another app while using your app.
- Does not consume valuable system resources when the user is not actively using it.

Does not lose the user's progress if they leave your app and return to it at a later time.
- Does not crash or lose the user's progress when the screen rotates between landscape and portrait orientation.

As you'll learn in the following lessons, there are several situations in which an activity transitions between different states that are illustrated in figure 1. However, only three of these states can be static. That is, the activity can exist in one of only three states for an extended period of time:

Resumed
: In this state, the activity is in the foreground and the user can interact with it. (Also sometimes referred to as the "running" state.)

Paused
: In this state, the activity is partially obscured by another activity—the other activity that's in the foreground is semi-transparent or doesn't cover the entire screen. The paused activity does not receive user input and cannot execute any code.

Stopped
: In this state, the activity is completely hidden and not visible to the user; it is considered to be in the background. While stopped, the activity instance and all its state information such as member variables is retained, but it cannot execute any code.

The other states (Created and Started) are transient and the system quickly moves from them to the next state by calling the next lifecycle callback method. That is, after the system calls `onCreate()`, it quickly calls `onStart()`, which is quickly followed by `onResume()`.

That's it for the basic activity lifecycle. Now you'll start learning about some of the specific lifecycle behaviors.

Specify Your App's Launcher Activity

When the user selects your app icon from the Home screen, the system calls the `onCreate()` method for the `Activity` in your app that you've declared to be the "launcher" (or "main") activity. This is the activity that serves as the main entry point to your app's user interface.

You can define which activity to use as the main activity in the Android manifest file, `AndroidManifest.xml`, which is at the root of your project directory.

The main activity for your app must be declared in the manifest with an `<intent-filter>` that includes the `MAIN` action and `LAUNCHER` category. For example:

```
<activity android:name=".MainActivity" android:label="@string/app_name">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

Note: When you create a new Android project with the Android SDK tools, the default project files include an `Activity` class that's declared in the manifest with this filter.

If either the `MAIN` action or `LAUNCHER` category are not declared for one of your activities, then your app icon will not appear in the Home screen's list of apps.

Create a New Instance

Most apps include several different activities that allow the user to perform different actions. Whether an activity is the main activity that's created when the user clicks your app icon or a different activity that your app starts in response to a user action, the system creates every new instance of `Activity` by calling its `onCreate()` method.

You must implement the `onCreate()` method to perform basic application startup logic that should happen only once for the entire life of the activity. For example, your implementation of `onCreate()` should define the user interface and possibly instantiate some class-scope variables.

For example, the following example of the `onCreate()` method shows some code that performs some fundamental setup for the activity, such as declaring the user interface (defined in an XML layout file), defining member variables, and configuring some of the UI.

```
TextView mTextView; // Member variable for text view in the layout

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    // Set the user interface layout for this Activity
    // The layout file is defined in the project
res/layout/main_activity.xml file
    setContentView(R.layout.main_activity);

    // Initialize member TextView so we can manipulate it later
    mTextView = (TextView) findViewById(R.id.text_message);
```

```
    // Make sure we're running on Honeycomb or higher to use ActionBar APIs
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
        // For the main activity, make sure the app icon in the action bar
        // does not behave as a button
        ActionBar actionBar = getActionBar();
        actionBar.setHomeButtonEnabled(false);
    }
}
```

Caution: Using the `SDK_INT` to prevent older systems from executing new APIs works in this way on Android 2.0 (API level 5) and higher only. Older versions will encounter a runtime exception.

Once the `onCreate()` finishes execution, the system calls the `onStart()` and `onResume()` methods in quick succession. Your activity never resides in the Created or Started states. Technically, the activity becomes visible to the user when `onStart()` is called, but `onResume()` quickly follows and the activity remains in the Resumed state until something occurs to change that, such as when a phone call is received, the user navigates to another activity, or the device screen turns off.

In the other lessons that follow, you'll see how the other start up methods, `onStart()` and `onResume()`, are useful during your activity's lifecycle when used to resume the activity from the Paused or Stopped states.

Note: The `onCreate()` method includes a parameter called `savedInstanceState` that's discussed in the latter lesson about Recreating an Activity.

Figure 2. Another illustration of the activity lifecycle structure with an emphasis

on the three main callbacks that the system calls in sequence when creating a new instance of the activity: `onCreate()`, `onStart()`, and `onResume()`. Once this sequence of callbacks complete, the activity reaches the Resumed state where users can interact with the activity until they switch to a different activity.

Destroy the Activity

While the activity's first lifecycle callback is `onCreate()`, its very last callback is `onDestroy()`. The system calls this method on your activity as the final signal that your activity instance is being completely removed from the system memory.

Most apps don't need to implement this method because local class references are destroyed with the activity and your activity should perform most cleanup during `onPause()` and `onStop()`. However, if your activity includes background threads that you created during `onCreate()` or other long-running resources that could potentially leak memory if not properly closed, you should kill them during `onDestroy()`.

```
@Override
public void onDestroy() {
    super.onDestroy();  // Always call the superclass

    // Stop method tracing that the activity started during onCreate()
    android.os.Debug.stopMethodTracing();
}
```

Note: The system calls `onDestroy()` after it has already called `onPause()` and `onStop()` in all situations except one: when you call `finish()` from within the `onCreate()` method. In some cases, such as when your activity operates as a temporary decision maker to launch another activity, you might call `finish()` from within `onCreate()` to destroy the activity. In this case, the system immediately calls `onDestroy()` without calling any of the other lifecycle methods.

Next Get started Next class

Pausing and Resuming an Activity

Previous Next Get started

This lesson teaches you to

1. Pause Your Activity
2. Resume Your Activity

You should also read

- Activities

Try it out

Download the demo

ActivityLifecycle.zip

During normal app use, the foreground activity is sometimes obstructed by other visual components that cause the activity to *pause*. For example, when a semi-transparent activity opens (such as one in the style of a dialog), the previous activity pauses. As long as the activity is still partially visible but currently not the activity in focus, it remains paused.

However, once the activity is fully-obstructed and not visible, it *stops* (which is discussed in the next lesson).

As your activity enters the paused state, the system calls the `onPause()` method on your `Activity`, which allows you to stop ongoing actions that should not continue while paused (such as a video) or persist any information that should be permanently saved in case the user continues to leave your app. If the user returns to your activity from the paused state, the system resumes it and calls the `onResume()` method.

Note: When your activity receives a call to `onPause()`, it may be an indication that the activity will be paused for a moment and the user may return focus to your activity. However, it's usually the first indication that the user is leaving your activity.

Figure 1. When a semi-transparent activity obscures your activity, the system calls `onPause()` and the activity waits in the Paused state (1). If the user returns to the activity while it's still paused, the system calls `onResume()` (2).

Pause Your Activity

When the system calls `onPause()` for your activity, it technically means your activity is still partially visible, but most often is an indication that the user is leaving the activity and it will soon enter the Stopped state. You should usually use the `onPause()` callback to:

- Stop animations or other ongoing actions that could consume CPU.
- Commit unsaved changes, but only if users expect such changes to be permanently saved when they leave (such as a draft email).
- Release system resources, such as broadcast receivers, handles to sensors (like GPS), or any resources that may affect battery life while your activity is paused and the user does not need them.

For example, if your application uses the `Camera`, the `onPause()` method is a good place to release it.

```
@Override
public void onPause() {
    super.onPause();  // Always call the superclass method first

    // Release the Camera because we don't need it when paused
    // and other activities might need to use it.
    if (mCamera != null) {
        mCamera.release()
        mCamera = null;
    }
}
```

Generally, you should **not** use `onPause()` to store user changes (such as personal information entered into a form) to permanent storage. The only time you should persist user changes to permanent storage within `onPause()` is when you're certain users expect the changes to be auto-saved (such as when drafting an email). However, you should avoid performing CPU-intensive work during `onPause()`, such as writing to a database, because it can slow the visible transition to the next activity (you should instead perform heavy-load shutdown operations during `onStop()`).

You should keep the amount of operations done in the `onPause()` method relatively simple in order to allow for a speedy transition to the user's next destination if your activity is actually being stopped.

Note: When your activity is paused, the `Activity` instance is kept resident in memory and is recalled when the activity resumes. You don't need to re-initialize components that were created during any of the callback methods leading up to the Resumed state.

Resume Your Activity

When the user resumes your activity from the Paused state, the system calls the `onResume()` method.

Be aware that the system calls this method every time your activity comes into the foreground, including when it's created for the first time. As such, you should implement `onResume()` to initialize components that you release during `onPause()` and perform any other initializations that must occur each time the activity enters the Resumed state (such as begin animations and

initialize components only used while the activity has user focus).

The following example of `onResume()` is the counterpart to the `onPause()` example above, so it initializes the camera that's released when the activity pauses.

```
@Override
public void onResume() {
    super.onResume();  // Always call the superclass method first

    // Get the Camera instance as the activity achieves full user focus
    if (mCamera == null) {
        initializeCamera(); // Local method to handle camera init
    }
}
```

Next Get started Next class

Stopping and Restarting an Activity

Previous Next Get started

This lesson teaches you to

1. Stop Your Activity
2. Start/Restart Your Activity

You should also read

- Activities

Try it out

Download the demo

ActivityLifecycle.zip

Properly stopping and restarting your activity is an important process in the activity lifecycle that ensures your users perceive that your app is always alive and doesn't lose their progress. There are a few of key scenarios in which your activity is stopped and restarted:

- The user opens the Recent Apps window and switches from your app to another app. The activity in your app that's currently in the foreground is stopped. If the user returns to your app from the Home screen launcher icon or the Recent Apps window, the activity restarts.
- The user performs an action in your app that starts a new activity. The current activity is stopped when the second activity is created. If the user then presses the *Back* button, the first activity is restarted.
- The user receives a phone call while using your app on his or her phone.

The `Activity` class provides two lifecycle methods, `onStop()` and `onRestart()`, which allow you to specifically handle how your activity handles being stopped and restarted. Unlike the paused state, which identifies a partial UI obstruction, the stopped state guarantees that the UI is no longer visible and the user's focus is in a separate activity (or an entirely separate app).

Note: Because the system retains your `Activity` instance in system memory when it is stopped, it's possible that you don't need to implement the `onStop()` and `onRestart()` (or even `onStart()` methods at all. For most activities that are relatively simple, the activity will stop and restart just fine and you might only need to use `onPause()` to pause ongoing actions and disconnect from system resources.

Figure 1. When the user leaves your activity, the system calls `onStop()` to stop the activity (1). If the user returns while the activity is stopped, the system calls `onRestart()` (2), quickly followed by `onStart()` (3) and `onResume()` (4). Notice that no matter what scenario causes the activity to stop, the system always calls `onPause()` before calling `onStop()`.

Stop Your Activity

When your activity receives a call to the `onStop()` method, it's no longer visible and should release almost all resources that aren't needed while the

user is not using it. Once your activity is stopped, the system might destroy the instance if it needs to recover system memory. In extreme cases, the system might simply kill your app process without calling the activity's final `onDestroy()` callback, so it's important you use `onStop()` to release resources that might leak memory.

Although the `onPause()` method is called before `onStop()`, you should use `onStop()` to perform larger, more CPU intensive shut-down operations, such as writing information to a database.

For example, here's an implementation of `onStop()` that saves the contents of a draft note to persistent storage:

```
@Override
protected void onStop() {
    super.onStop();  // Always call the superclass method first

    // Save the note's current draft, because the activity is stopping
    // and we want to be sure the current note progress isn't lost.
    ContentValues values = new ContentValues();
    values.put(NotePad.Notes.COLUMN_NAME_NOTE, getCurrentNoteText());
    values.put(NotePad.Notes.COLUMN_NAME_TITLE, getCurrentNoteTitle());

    getContentResolver().update(
            mUri,    // The URI for the note to update.
            values,  // The map of column names and new values to apply
to them.
            null,    // No SELECT criteria are used.
            null     // No WHERE columns are used.
            );
}
```

When your activity is stopped, the `Activity` object is kept resident in memory and is recalled when the activity resumes. You don’t need to re-initialize components that were created during any of the callback methods leading up to the Resumed state. The system also keeps track of the current state for each `View` in the layout, so if the user entered text into an `EditText` widget, that content is retained so you don't need to save and restore it.

Note: Even if the system destroys your activity while it's stopped, it still retains the state of the `View` objects (such as text in an `EditText`) in a `Bundle` (a blob of key-value pairs) and restores them if the user navigates back to the same instance of the activity (the next lesson talks more about using a `Bundle` to save other state data in case your activity is destroyed and recreated).

Start/Restart Your Activity

When your activity comes back to the foreground from the stopped state, it receives a call to `onRestart()`. The system also calls the `onStart()` method, which happens every time your activity becomes visible (whether being restarted or created for the first time). The `onRestart()` method, however, is called only when the activity resumes from the stopped state, so you can use it to perform special restoration work that might be necessary only if the activity was previously stopped, but not destroyed.

It's uncommon that an app needs to use `onRestart()` to restore the activity's state, so there aren't any guidelines for this method that apply to the general population of apps. However, because your `onStop()` method should essentially clean up all your activity's resources, you'll need to re-instantiate them when the activity restarts. Yet, you also need to instantiate them when your activity is created for the first time (when there's no existing instance of the activity). For this reason, you should usually use the `onStart()` callback method as the counterpart to the `onStop()` method, because the system calls `onStart()` both when it creates your activity and when it restarts the activity from the stopped state.

For example, because the user might have been away from your app for a long time before coming back it, the `onStart()` method is a good place to verify that required system features are enabled:

```
@Override
protected void onStart() {
    super.onStart();  // Always call the superclass method first

    // The activity is either being restarted or started for the first
time
    // so this is where we should make sure that GPS is enabled
    LocationManager locationManager =
            (LocationManager)
getSystemService(Context.LOCATION_SERVICE);
    boolean gpsEnabled =
locationManager.isProviderEnabled(LocationManager.GPS_PROVIDER);

    if (!gpsEnabled) {
        // Create a dialog here that requests the user to enable GPS,
and use an intent
        // with the
android.provider.Settings.ACTION_LOCATION_SOURCE_SETTINGS action
        // to take the user to the Settings screen to enable GPS when
they click "OK"
    }
}
```

```
@Override
protected void onRestart() {
    super.onRestart();  // Always call the superclass method first

    // Activity being restarted from stopped state
}
```

When the system destroys your activity, it calls the `onDestroy()` method for your `Activity`. Because you should generally have released most of your resources with `onStop()`, by the time you receive a call to `onDestroy()`, there's not much that most apps need to do. This method is your last chance to clean out resources that could lead to a memory leak, so you should be sure that additional threads are destroyed and other long-running actions like method tracing are also stopped.

Next Get started Next class

Recreating an Activity

Previous Next Get started

This lesson teaches you to

1. Save Your Activity State
2. Restore Your Activity State

You should also read

- Supporting Different Screens
- Handling Runtime Changes
- Activities

There are a few scenarios in which your activity is destroyed due to normal app behavior, such as when the user presses the *Back* button or your activity signals its own destruction by calling `finish()`. The system may also destroy your activity if it's currently stopped and hasn't been used in a long time or the foreground activity requires more resources so the system must shut down background processes to recover memory.

When your activity is destroyed because the user presses *Back* or the activity finishes itself, the system's concept of that `Activity` instance is gone forever because the behavior indicates the activity is no longer needed. However, if the system destroys the activity due to system constraints (rather than normal app behavior), then although the actual `Activity` instance is gone, the system remembers that it existed such that if the user navigates back to it, the system creates a new instance of the activity using a set of saved data that describes the state of the activity when it was destroyed. The saved data that the system uses to restore the previous state is called the "instance state" and is a collection of key-value pairs stored in a `Bundle` object.

Caution: Your activity will be destroyed and recreated each time the user rotates the screen. When the screen changes orientation, the system destroys and recreates the foreground activity because the screen configuration has changed and your activity might need to load alternative resources (such as the layout).

By default, the system uses the `Bundle` instance state to save information about each `View` object in your activity layout (such as the text value entered into an `EditText` object). So, if your activity instance is destroyed and recreated, the state of the layout is restored to its previous state with no code required by you. However, your activity might have more state information that you'd like to restore, such as member variables that track the user's progress in the activity.

Note: In order for the Android system to restore the state of the views in your activity, **each view must have a unique ID**, supplied by the `android:id` attribute.

To save additional data about the activity state, you must override the `onSaveInstanceState()` callback method. The system calls this method when the user is leaving your activity and passes it the `Bundle` object that will be saved in the event that your activity is destroyed unexpectedly. If the system must recreate the activity instance later, it passes the same `Bundle` object to both the `onRestoreInstanceState()` and `onCreate()` methods.

Figure 2. As the system begins to stop your activity, it calls

onSaveInstanceState() (1) so you can specify additional state data you'd like to save in case the Activity instance must be recreated. If the activity is destroyed and the same instance must be recreated, the system passes the state data defined at (1) to both the onCreate() method (2) and the onRestoreInstanceState() method (3).

Save Your Activity State

As your activity begins to stop, the system calls onSaveInstanceState() so your activity can save state information with a collection of key-value pairs. The default implementation of this method saves information about the state of the activity's view hierarchy, such as the text in an EditText widget or the scroll position of a ListView.

To save additional state information for your activity, you must implement onSaveInstanceState() and add key-value pairs to the Bundle object. For example:

```
static final String STATE_SCORE = "playerScore";
static final String STATE_LEVEL = "playerLevel";
...

@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
    // Save the user's current game state
    savedInstanceState.putInt(STATE_SCORE, mCurrentScore);
    savedInstanceState.putInt(STATE_LEVEL, mCurrentLevel);

    // Always call the superclass so it can save the view hierarchy
state
    super.onSaveInstanceState(savedInstanceState);
}
```

Caution: Always call the superclass implementation of onSaveInstanceState() so the default implementation can save the state of the view hierarchy.

Restore Your Activity State

When your activity is recreated after it was previously destroyed, you can recover your saved state from the Bundle that the system passes your activity. Both the onCreate() and onRestoreInstanceState() callback

methods receive the same Bundle that contains the instance state information.

Because the onCreate() method is called whether the system is creating a new instance of your activity or recreating a previous one, you must check whether the state Bundle is null before you attempt to read it. If it is null, then the system is creating a new instance of the activity, instead of restoring a previous one that was destroyed.

For example, here's how you can restore some state data in onCreate():

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState); // Always call the
superclass first

    // Check whether we're recreating a previously destroyed
instance
    if (savedInstanceState != null) {
        // Restore value of members from saved state
        mCurrentScore = savedInstanceState.getInt(STATE_SCORE);
        mCurrentLevel = savedInstanceState.getInt(STATE_LEVEL);
    } else {
        // Probably initialize members with default values for a new
instance
    }
    ...
}
```

Instead of restoring the state during onCreate() you may choose to implement onRestoreInstanceState(), which the system calls after the onStart() method. The system calls onRestoreInstanceState() only if there is a saved state to restore, so you do not need to check whether the Bundle is null:

```
public void onRestoreInstanceState(Bundle savedInstanceState) {
    // Always call the superclass so it can restore the view
hierarchy
    super.onRestoreInstanceState(savedInstanceState);

    // Restore state members from saved instance
    mCurrentScore = savedInstanceState.getInt(STATE_SCORE);
    mCurrentLevel = savedInstanceState.getInt(STATE_LEVEL);
}
```

Caution: Always call the superclass implementation of onRestoreInstanceState() so the default implementation can restore the state of the view hierarchy.

To learn more about recreating your activity due to a restart event at runtime (such as when the screen rotates), read Handling Runtime Changes.

Next Get started Next class

Building a Dynamic UI with Fragments

Previous Next Get started

Dependencies and prerequisites

- Basic knowledge of the Activity lifecycle (see Managing the Activity Lifecycle)
- Experience building XML layouts

You should also read

- Fragments
- Supporting Tablets and Handsets

Try it out

Download the sample

FragmentBasics.zip

To create a dynamic and multi-pane user interface on Android, you need to encapsulate UI components and activity behaviors into modules that you can swap into and out of your activities. You can create these modules with the `Fragment` class, which behaves somewhat like a nested activity that can define its own layout and manage its own lifecycle.

When a fragment specifies its own layout, it can be configured in different combinations with other fragments inside an activity to modify your layout configuration for different screen sizes (a small screen might show one fragment at a time, but a large screen can show two or more).

This class shows you how to create a dynamic user experience with fragments and optimize your app's user experience for devices with different screen sizes, all while continuing to support devices running versions as old as Android 1.6.

Lessons

Creating a Fragment

Learn how to build a fragment and implement basic behaviors within its callback methods.

Building a Flexible UI

Learn how to build your app with layouts that provide different fragment configurations for different screens.

Communicating with Other Fragments

Learn how to set up communication paths from a fragment to the activity and other fragments.

Next Get started Next class

Creating a Fragment

Previous Next Get started

This lesson teaches you to

1. Create a Fragment Class
2. Add a Fragment to an Activity using XML

You should also read

- Fragments

Try it out

Download the sample

FragmentBasics.zip

You can think of a fragment as a modular section of an activity, which has its own lifecycle, receives its own input events, and which you can add or remove while the activity is running (sort of like a "sub activity" that you can reuse in different activities). This lesson shows how to extend the `Fragment` class using the Support Library so your app remains compatible with devices running system versions as low as Android 1.6.

Note: If you decide that the minimum API level your app requires is 11 or higher, you don't need to use the Support Library and can instead use the framework's built in `Fragment` class and related APIs. Just be aware that this lesson is focused on using the APIs from the Support Library, which use a specific package signature and sometimes slightly different API names than the versions included in the platform.

Before you begin this lesson, you must set up your Android project to use the Support Library. If you have not used the Support Library before, set up your project to use the **v4** library by following the Support Library Setup document. However, you can also include the action bar in your activities by instead using the **v7 appcompat** library, which is compatible with Android 2.1 (API level 7) and also includes the `Fragment` APIs.

Create a Fragment Class

To create a fragment, extend the `Fragment` class, then override key lifecycle methods to insert your app logic, similar to the way you would with an `Activity` class.

One difference when creating a `Fragment` is that you must use the `onCreateView()` callback to define the layout. In fact, this is the only callback you need in order to get a fragment running. For example, here's a simple fragment that specifies its own layout:

```
import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.ViewGroup;

public class ArticleFragment extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup
container,
        Bundle savedInstanceState) {
        // Inflate the layout for this fragment
        return inflater.inflate(R.layout.article_view, container,
false);
    }
}
```

Just like an activity, a fragment should implement other lifecycle callbacks that allow you to manage its state as it is added or removed from the activity and as the activity transitions between its lifecycle states. For instance, when the activity's `onPause()` method is called, any fragments in the activity also receive a call to `onPause()`.

More information about the fragment lifecycle and callback methods is available in the Fragments developer guide.

Add a Fragment to an Activity using XML

While fragments are reusable, modular UI components, each instance of a `Fragment` class must be associated with a parent `FragmentActivity`. You can achieve this association by defining each fragment within your activity layout XML file.

Note: `FragmentActivity` is a special activity provided in the Support Library to handle fragments on system versions older than API level 11. If the lowest system version you support is API level 11 or higher, then you can use a regular `Activity`.

Here is an example layout file that adds two fragments to an activity when the device screen is considered "large" (specified by the `large` qualifier in the directory name).

res/layout-large/news_articles.xml

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <fragment
android:name="com.example.android.fragments.HeadlinesFragment"
              android:id="@+id/headlines_fragment"
              android:layout_weight="1"
              android:layout_width="0dp"
              android:layout_height="match_parent" />

    <fragment
android:name="com.example.android.fragments.ArticleFragment"
              android:id="@+id/article_fragment"
              android:layout_weight="2"
              android:layout_width="0dp"
              android:layout_height="match_parent" />

</LinearLayout>
```

Tip: For more about creating layouts for different screen sizes, read Supporting Different Screen Sizes.

Then apply the layout to your activity:

```
import android.os.Bundle;
import android.support.v4.app.FragmentActivity;
```

```
public class MainActivity extends FragmentActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.news_articles);
    }
}
```

If you're using the v7 appcompat library, your activity should instead extend `ActionBarActivity`, which is a subclass of `FragmentActivity` (for more information, read Adding the Action Bar).

Note: When you add a fragment to an activity layout by defining the fragment in the layout XML file, you *cannot* remove the fragment at runtime. If you plan to swap your fragments in and out during user interaction, you must add the fragment to the activity when the activity first starts, as shown in the next lesson.

Next Get started Next class

Building a Flexible UI

Previous Next Get started

This lesson teaches you to

1. Add a Fragment to an Activity at Runtime
2. Replace One Fragment with Another

You should also read

- Fragments
- Supporting Tablets and Handsets

Try it out

Download the sample

FragmentBasics.zip

When designing your application to support a wide range of screen sizes, you can reuse your fragments in different layout configurations to optimize the user experience based on the available screen space.

For example, on a handset device it might be appropriate to display just one fragment at a time for a single-pane user interface. Conversely, you may want to set fragments side-by-side on a tablet which has a wider screen size to display more information to the user.

Figure 1. Two fragments, displayed in different configurations for the same activity on different screen sizes. On a large screen, both fragments fit side by side, but on a handset device, only one fragment fits at a time so the fragments must replace each other as the user navigates.

The `FragmentManager` class provides methods that allow you to add, remove, and replace fragments to an activity at runtime in order to create a dynamic experience.

Add a Fragment to an Activity at Runtime

Rather than defining the fragments for an activity in the layout file—as shown in the previous lesson with the `<fragment>` element—you can add a fragment to the activity during the activity runtime. This is necessary if you plan to change fragments during the life of the activity.

To perform a transaction such as add or remove a fragment, you must use the `FragmentManager` to create a `FragmentTransaction`, which provides APIs to add, remove, replace, and perform other fragment transactions.

If your activity allows the fragments to be removed and replaced, you should add the initial fragment(s) to the activity during the activity's `onCreate()` method.

An important rule when dealing with fragments—especially when adding fragments at runtime—is that your activity layout must include a

container View in which you can insert the fragment.

The following layout is an alternative to the layout shown in the previous lesson that shows only one fragment at a time. In order to replace one fragment with another, the activity's layout includes an empty FrameLayout that acts as the fragment container.

Notice that the filename is the same as the layout file in the previous lesson, but the layout directory does *not* have the `large` qualifier, so this layout is used when the device screen is smaller than *large* because the screen does not fit both fragments at the same time.

```
res/layout/news_articles.xml:

<FrameLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/fragment_container"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

Inside your activity, call getSupportFragmentManager() to get a FragmentManager using the Support Library APIs. Then call beginTransaction() to create a FragmentTransaction and call add() to add a fragment.

You can perform multiple fragment transaction for the activity using the same FragmentTransaction. When you're ready to make the changes, you must call commit().

For example, here's how to add a fragment to the previous layout:

```
import android.os.Bundle;
import android.support.v4.app.FragmentActivity;

public class MainActivity extends FragmentActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.news_articles);

        // Check that the activity is using the layout version with
        // the fragment_container FrameLayout
        if (findViewById(R.id.fragment_container) != null) {

            // However, if we're being restored from a previous
state,
            // then we don't need to do anything and should return
```

```
or else
            // we could end up with overlapping fragments.
            if (savedInstanceState != null) {
                return;
            }

            // Create a new Fragment to be placed in the activity
layout
            HeadlinesFragment firstFragment = new
HeadlinesFragment();

            // In case this activity was started with special
instructions from an
            // Intent, pass the Intent's extras to the fragment as
arguments
            firstFragment.setArguments(getIntent().getExtras());

            // Add the fragment to the 'fragment_container'
FrameLayout
            getSupportFragmentManager().beginTransaction()
                    .add(R.id.fragment_container,
firstFragment).commit();
        }
    }
}
```

Because the fragment has been added to the `FrameLayout` container at runtime—instead of defining it in the activity's layout with a `<fragment>` element—the activity can remove the fragment and replace it with a different one.

Replace One Fragment with Another

The procedure to replace a fragment is similar to adding one, but requires the `replace()` method instead of `add()`.

Keep in mind that when you perform fragment transactions, such as replace or remove one, it's often appropriate to allow the user to navigate backward and "undo" the change. To allow the user to navigate backward through the fragment transactions, you must call `addToBackStack()` before you commit the `FragmentTransaction`.

Note: When you remove or replace a fragment and add the transaction to the back stack, the fragment that is removed is stopped (not destroyed). If the user navigates back to restore the fragment, it restarts. If you *do not* add the transaction to the back stack, then the fragment is destroyed when removed or replaced.

Example of replacing one fragment with another:

```
// Create fragment and give it an argument specifying the article it
should show
ArticleFragment newFragment = new ArticleFragment();
Bundle args = new Bundle();
args.putInt(ArticleFragment.ARG_POSITION, position);
newFragment.setArguments(args);

FragmentTransaction transaction =
getSupportFragmentManager().beginTransaction();

// Replace whatever is in the fragment_container view with this
fragment,
// and add the transaction to the back stack so the user can
navigate back
transaction.replace(R.id.fragment_container, newFragment);
transaction.addToBackStack(null);

// Commit the transaction
transaction.commit();
```

The `addToBackStack()` method takes an optional string parameter that specifies a unique name for the transaction. The name isn't needed unless you plan to perform advanced fragment operations using the `FragmentManager.BackStackEntry` APIs.

Next Get started Next class

Communicating with Other Fragments

Previous Next Get started

This lesson teaches you to

1. Define an Interface
2. Implement the Interface
3. Deliver a Message to a Fragment

You should also read

- Fragments

Try it out

Download the sample

FragmentBasics.zip

In order to reuse the Fragment UI components, you should build each as a completely self-contained, modular component that defines its own layout and behavior. Once you have defined these reusable Fragments, you can associate them with an Activity and connect them with the application logic to realize the overall composite UI.

Often you will want one Fragment to communicate with another, for example to change the content based on a user event. All Fragment-to-Fragment communication is done through the associated Activity. Two Fragments should never communicate directly.

Define an Interface

To allow a Fragment to communicate up to its Activity, you can define an interface in the Fragment class and implement it within the Activity. The Fragment captures the interface implementation during its onAttach() lifecycle method and can then call the Interface methods in order to communicate with the Activity.

Here is an example of Fragment to Activity communication:

```
public class HeadlinesFragment extends ListFragment {
    OnHeadlineSelectedListener mCallback;

    // Container Activity must implement this interface
    public interface OnHeadlineSelectedListener {
        public void onArticleSelected(int position);
    }

    @Override
    public void onAttach(Activity activity) {
        super.onAttach(activity);

        // This makes sure that the container activity has
implemented
        // the callback interface. If not, it throws an exception
        try {
            mCallback = (OnHeadlineSelectedListener) activity;
        } catch (ClassCastException e) {
            throw new ClassCastException(activity.toString()
                    + " must implement OnHeadlineSelectedListener");
        }
    }

    ...
}
```

Now the fragment can deliver messages to the activity by calling the `onArticleSelected()` method (or other methods in the interface) using the `mCallback` instance of the `OnHeadlineSelectedListener` interface.

For example, the following method in the fragment is called when the user clicks on a list item. The fragment uses the callback interface to deliver the event to the parent activity.

```
    @Override
    public void onListItemClick(ListView l, View v, int position,
long id) {
        // Send the event to the host activity
        mCallback.onArticleSelected(position);
    }
```

Implement the Interface

In order to receive event callbacks from the fragment, the activity that hosts it must implement the interface defined in the fragment class.

For example, the following activity implements the interface from the above example.

```
public static class MainActivity extends Activity
        implements HeadlinesFragment.OnHeadlineSelectedListener{
    ...

    public void onArticleSelected(int position) {
        // The user selected the headline of an article from the
HeadlinesFragment
        // Do something here to display that article
    }
}
```

Deliver a Message to a Fragment

The host activity can deliver messages to a fragment by capturing the `Fragment` instance with `findFragmentById()`, then directly call the fragment's public methods.

For instance, imagine that the activity shown above may contain another fragment that's used to display the item specified by the data returned in the above callback method. In this case, the activity can pass the information received in the callback method to the other fragment that will display the item:

```
public static class MainActivity extends Activity
        implements HeadlinesFragment.OnHeadlineSelectedListener{
    ...

    public void onArticleSelected(int position) {
        // The user selected the headline of an article from the
HeadlinesFragment
        // Do something here to display that article

        ArticleFragment articleFrag = (ArticleFragment)

getSupportFragmentManager().findFragmentById(R.id.article_fragment);

        if (articleFrag != null) {
            // If article frag is available, we're in two-pane
```

```
layout...

            // Call a method in the ArticleFragment to update its
content
            articleFrag.updateArticleView(position);
        } else {
            // Otherwise, we're in the one-pane layout and must swap
frags...

            // Create fragment and give it an argument for the
selected article
            ArticleFragment newFragment = new ArticleFragment();
            Bundle args = new Bundle();
            args.putInt(ArticleFragment.ARG_POSITION, position);
            newFragment.setArguments(args);

            FragmentTransaction transaction =
getSupportFragmentManager().beginTransaction();

            // Replace whatever is in the fragment_container view
with this fragment,
            // and add the transaction to the back stack so the user
can navigate back
            transaction.replace(R.id.fragment_container,
newFragment);
            transaction.addToBackStack(null);

            // Commit the transaction
            transaction.commit();
        }
    }
}
```

Next Get started Next class

Saving Data

Previous Next Get started

Dependencies and prerequisites

- Android 1.6 (API Level 4) or higher
- Familiarity with Map key-value collections
- Familiarity with the Java file I/O API
- Familiarity with SQL databases

You should also read

- Storage Options

Most Android apps need to save data, even if only to save information about the app state during `onPause()` so the user's progress is not lost. Most non-trivial apps also need to save user settings, and some apps must manage large amounts of information in files and databases. This class introduces you to the principal data storage options in Android, including:

- Saving key-value pairs of simple data types in a shared preferences file
- Saving arbitrary files in Android's file system
- Using databases managed by SQLite

Lessons

Saving Key-Value Sets

Learn to use a shared preferences file for storing small amounts of information in key-value pairs.

Saving Files

Learn to save a basic file, such as to store long sequences of data that are generally read in order.

Saving Data in SQL Databases

Learn to use a SQLite database to read and write structured data.

Next Get started Next class

Saving Key-Value Sets

Previous Next Get started

This lesson teaches you to

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

You should also read

- Using Shared Preferences

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

This class shows you how to use the `SharedPreferences` APIs to store and retrieve simple values.

Note: The `SharedPreferences` APIs are only for reading and writing key-value pairs and you should not confuse them with the `Preference` APIs, which help you build a user interface for your app settings (although they use `SharedPreferences` as their implementation to save the app settings). For information about using the `Preference` APIs, see the Settings guide.

Get a Handle to a SharedPreferences

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

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

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

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

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

Alternatively, if you need just one shared preference file for your activity, you can use the `getPreferences()` method:

```
SharedPreferences sharedPref =
getActivity().getPreferences(Context.MODE_PRIVATE);
```

Caution: If you create a shared preferences file with `MODE_WORLD_READABLE` or `MODE_WORLD_WRITEABLE`, then any other apps that know the file identifier can access your data.

Write to Shared Preferences

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

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

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

Read from Shared Preferences

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

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

Next Get started Next class

Saving Files

Previous Next Get started

This lesson teaches you to

1. Choose Internal or External Storage
2. Obtain Permissions for External Storage
3. Save a File on Internal Storage
4. Save a File on External Storage
5. Query Free Space
6. Delete a File

You should also read

- Using the Internal Storage
- Using the External Storage

Android uses a file system that's similar to disk-based file systems on other platforms. This lesson describes how to work with the Android file system to read and write files with the `File` APIs.

A `File` object is suited to reading or writing large amounts of data in start-to-finish order without skipping around. For example, it's good for image files or anything exchanged over a network.

This lesson shows how to perform basic file-related tasks in your app. The lesson assumes that you are familiar with the basics of the Linux file system and the standard file input/output APIs in `java.io`.

Choose Internal or External Storage

All Android devices have two file storage areas: "internal" and "external"

storage. These names come from the early days of Android, when most devices offered built-in non-volatile memory (internal storage), plus a removable storage medium such as a micro SD card (external storage). Some devices divide the permanent storage space into "internal" and "external" partitions, so even without a removable storage medium, there are always two storage spaces and the API behavior is the same whether the external storage is removable or not. The following lists summarize the facts about each storage space.

Internal storage:

- It's always available.
- Files saved here are accessible by only your app by default.
- When the user uninstalls your app, the system removes all your app's files from internal storage.

Internal storage is best when you want to be sure that neither the user nor other apps can access your files.

External storage:

- It's not always available, because the user can mount the external storage as USB storage and in some cases remove it from the device.
- It's world-readable, so files saved here may be read outside of your control.
- When the user uninstalls your app, the system removes your app's files from here only if you save them in the directory from `getExternalFilesDir()`.

External storage is the best place for files that don't require access restrictions and for files that you want to share with other apps or allow the user to access with a computer.

Tip: Although apps are installed onto the internal storage by default, you can specify the `android:installLocation` attribute in your manifest so your app may be installed on external storage. Users appreciate this option when the APK size is very large and they have an external storage

space that's larger than the internal storage. For more information, see App Install Location.

Obtain Permissions for External Storage

To write to the external storage, you must request the `WRITE_EXTERNAL_STORAGE` permission in your manifest file:

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

Caution: Currently, all apps have the ability to read the external storage without a special permission. However, this will change in a future release. If your app needs to read the external storage (but not write to it), then you will need to declare the `READ_EXTERNAL_STORAGE` permission. To ensure that your app continues to work as expected, you should declare this permission now, before the change takes effect.

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

However, if your app uses the `WRITE_EXTERNAL_STORAGE` permission, then it implicitly has permission to read the external storage as well.

You don't need any permissions to save files on the internal storage. Your application always has permission to read and write files in its internal storage directory.

Save a File on Internal Storage

When saving a file to internal storage, you can acquire the appropriate directory as a `File` by calling one of two methods:

`getFilesDir()`
: Returns a `File` representing an internal directory for your app.

getCacheDir()

Returns a File representing an internal directory for your app's temporary cache files. Be sure to delete each file once it is no longer needed and implement a reasonable size limit for the amount of memory you use at any given time, such as 1MB. If the system begins running low on storage, it may delete your cache files without warning.

To create a new file in one of these directories, you can use the File() constructor, passing the File provided by one of the above methods that specifies your internal storage directory. For example:

```
File file = new File(context.getFilesDir(), filename);
```

Alternatively, you can call openFileOutput() to get a FileOutputStream that writes to a file in your internal directory. For example, here's how to write some text to a file:

```
String filename = "myfile";
String string = "Hello world!";
FileOutputStream outputStream;

try {
  outputStream = openFileOutput(filename, Context.MODE_PRIVATE);
  outputStream.write(string.getBytes());
  outputStream.close();
} catch (Exception e) {
  e.printStackTrace();
}
```

Or, if you need to cache some files, you should instead use createTempFile(). For example, the following method extracts the file name from a URL and creates a file with that name in your app's internal cache directory:

```
public File getTempFile(Context context, String url) {
    File file;
    try {
        String fileName = Uri.parse(url).getLastPathSegment();
        file = File.createTempFile(fileName, null,
context.getCacheDir());
    catch (IOException e) {
        // Error while creating file
    }
    return file;
}
```

Note: Your app's internal storage directory is specified by your app's

package name in a special location of the Android file system. Technically, another app can read your internal files if you set the file mode to be readable. However, the other app would also need to know your app package name and file names. Other apps cannot browse your internal directories and do not have read or write access unless you explicitly set the files to be readable or writable. So as long as you use MODE_PRIVATE for your files on the internal storage, they are never accessible to other apps.

Save a File on External Storage

Because the external storage may be unavailable—such as when the user has mounted the storage to a PC or has removed the SD card that provides the external storage—you should always verify that the volume is available before accessing it. You can query the external storage state by calling getExternalStorageState(). If the returned state is equal to MEDIA_MOUNTED, then you can read and write your files. For example, the following methods are useful to determine the storage availability:

```
/* Checks if external storage is available for read and write */
public boolean isExternalStorageWritable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state)) {
        return true;
    }
    return false;
}

/* Checks if external storage is available to at least read */
public boolean isExternalStorageReadable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state) ||
        Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
        return true;
    }
    return false;
}
```

Although the external storage is modifiable by the user and other apps, there are two categories of files you might save here:

Public files

Files that should be freely available to other apps and to the user. When the user uninstalls your app, these files should remain

available to the user.

For example, photos captured by your app or other downloaded files.

Private files

Files that rightfully belong to your app and should be deleted when the user uninstalls your app. Although these files are technically accessible by the user and other apps because they are on the external storage, they are files that realistically don't provide value to the user outside your app. When the user uninstalls your app, the system deletes all files in your app's external private directory.

For example, additional resources downloaded by your app or temporary media files.

If you want to save public files on the external storage, use the `getExternalStoragePublicDirectory()` method to get a `File` representing the appropriate directory on the external storage. The method takes an argument specifying the type of file you want to save so that they can be logically organized with other public files, such as `DIRECTORY_MUSIC` or `DIRECTORY_PICTURES`. For example:

```
public File getAlbumStorageDir(String albumName) {
    // Get the directory for the user's public pictures directory.
    File file = new
File(Environment.getExternalStoragePublicDirectory(
            Environment.DIRECTORY_PICTURES), albumName);
    if (!file.mkdirs()) {
        Log.e(LOG_TAG, "Directory not created");
    }
    return file;
}
```

If you want to save files that are private to your app, you can acquire the appropriate directory by calling `getExternalFilesDir()` and passing it a name indicating the type of directory you'd like. Each directory created this way is added to a parent directory that encapsulates all your app's external storage files, which the system deletes when the user uninstalls your app.

For example, here's a method you can use to create a directory for an

individual photo album:

```
public File getAlbumStorageDir(Context context, String albumName) {
    // Get the directory for the app's private pictures directory.
    File file = new File(context.getExternalFilesDir(
            Environment.DIRECTORY_PICTURES), albumName);
    if (!file.mkdirs()) {
        Log.e(LOG_TAG, "Directory not created");
    }
    return file;
}
```

If none of the pre-defined sub-directory names suit your files, you can instead call `getExternalFilesDir()` and pass `null`. This returns the root directory for your app's private directory on the external storage.

Remember that `getExternalFilesDir()` creates a directory inside a directory that is deleted when the user uninstalls your app. If the files you're saving should remain available after the user uninstalls your app—such as when your app is a camera and the user will want to keep the photos—you should instead use `getExternalStoragePublicDirectory()`.

Regardless of whether you use `getExternalStoragePublicDirectory()` for files that are shared or `getExternalFilesDir()` for files that are private to your app, it's important that you use directory names provided by API constants like `DIRECTORY_PICTURES`. These directory names ensure that the files are treated properly by the system. For instance, files saved in `DIRECTORY_RINGTONES` are categorized by the system media scanner as ringtones instead of music.

Query Free Space

If you know ahead of time how much data you're saving, you can find out whether sufficient space is available without causing an `IOException` by calling `getFreeSpace()` or `getTotalSpace()`. These methods provide the current available space and the total space in the storage volume, respectively. This information is also useful to avoid filling the storage volume above a certain threshold.

However, the system does not guarantee that you can write as many bytes as are indicated by `getFreeSpace()`. If the number returned is a few

MB more than the size of the data you want to save, or if the file system is less than 90% full, then it's probably safe to proceed. Otherwise, you probably shouldn't write to storage.

Note: You aren't required to check the amount of available space before you save your file. You can instead try writing the file right away, then catch an `IOException` if one occurs. You may need to do this if you don't know exactly how much space you need. For example, if you change the file's encoding before you save it by converting a PNG image to JPEG, you won't know the file's size beforehand.

Delete a File

You should always delete files that you no longer need. The most straightforward way to delete a file is to have the opened file reference call `delete()` on itself.

```
myFile.delete();
```

If the file is saved on internal storage, you can also ask the `Context` to locate and delete a file by calling `deleteFile()`:

```
myContext.deleteFile(fileName);
```

Note: When the user uninstalls your app, the Android system deletes the following:

- All files you saved on internal storage
- All files you saved on external storage using `getExternalFilesDir()`.

However, you should manually delete all cached files created with `getCacheDir()` on a regular basis and also regularly delete other files you no longer need.

Next Get started Next class

Saving Data in SQL Databases

Previous Next Get started

This lesson teaches you to

1. Define a Schema and Contract
2. Create a Database Using a SQL Helper
3. Put Information into a Database
4. Read Information from a Database
5. Delete Information from a Database
6. Update a Database

You should also read

- Using Databases

Saving data to a database is ideal for repeating or structured data, such as contact information. This class assumes that you are familiar with SQL databases in general and helps you get started with SQLite databases on Android. The APIs you'll need to use a database on Android are available in the `android.database.sqlite` package.

Define a Schema and Contract

One of the main principles of SQL databases is the schema: a formal declaration of how the database is organized. The schema is reflected in the SQL statements that you use to create your database. You may find it helpful to create a companion class, known as a *contract* class, which explicitly specifies the layout of your schema in a systematic and self-documenting way.

A contract class is a container for constants that define names for URIs,

tables, and columns. The contract class allows you to use the same constants across all the other classes in the same package. This lets you change a column name in one place and have it propagate throughout your code.

A good way to organize a contract class is to put definitions that are global to your whole database in the root level of the class. Then create an inner class for each table that enumerates its columns.

Note: By implementing the `BaseColumns` interface, your inner class can inherit a primary key field called `_ID` that some Android classes such as cursor adaptors will expect it to have. It's not required, but this can help your database work harmoniously with the Android framework.

For example, this snippet defines the table name and column names for a single table:

```
public final class FeedReaderContract {
    // To prevent someone from accidentally instantiating the
contract class,
    // give it an empty constructor.
    public FeedReaderContract() {}

    /* Inner class that defines the table contents */
    public static abstract class FeedEntry implements BaseColumns {
        public static final String TABLE_NAME = "entry";
        public static final String COLUMN_NAME_ENTRY_ID = "entryid";
        public static final String COLUMN_NAME_TITLE = "title";
        public static final String COLUMN_NAME_SUBTITLE =
"subtitle";
        ...
    }
}
```

Create a Database Using a SQL Helper

Once you have defined how your database looks, you should implement methods that create and maintain the database and tables. Here are some typical statements that create and delete a table:

```
private static final String TEXT_TYPE = " TEXT";
private static final String COMMA_SEP = ",";
private static final String SQL_CREATE_ENTRIES =
    "CREATE TABLE " + FeedEntry.TABLE_NAME + " (" +
    FeedEntry._ID + " INTEGER PRIMARY KEY," +
    FeedEntry.COLUMN_NAME_ENTRY_ID + TEXT_TYPE + COMMA_SEP +
```

```
    FeedEntry.COLUMN_NAME_TITLE + TEXT_TYPE + COMMA_SEP +
    ... // Any other options for the CREATE command
    " )";

private static final String SQL_DELETE_ENTRIES =
    "DROP TABLE IF EXISTS " + FeedEntry.TABLE_NAME;
```

Just like files that you save on the device's internal storage, Android stores your database in private disk space that's associated application. Your data is secure, because by default this area is not accessible to other applications.

A useful set of APIs is available in the `SQLiteOpenHelper` class. When you use this class to obtain references to your database, the system performs the potentially long-running operations of creating and updating the database only when needed and *not during app startup*. All you need to do is call `getWritableDatabase()` or `getReadableDatabase()`.

Note: Because they can be long-running, be sure that you call `getWritableDatabase()` or `getReadableDatabase()` in a background thread, such as with `AsyncTask` or `IntentService`.

To use `SQLiteOpenHelper`, create a subclass that overrides the `onCreate()`, `onUpgrade()` and `onOpen()` callback methods. You may also want to implement `onDowngrade()`, but it's not required.

For example, here's an implementation of `SQLiteOpenHelper` that uses some of the commands shown above:

```
public class FeedReaderDbHelper extends SQLiteOpenHelper {
    // If you change the database schema, you must increment the
database version.
    public static final int DATABASE_VERSION = 1;
    public static final String DATABASE_NAME = "FeedReader.db";

    public FeedReaderDbHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(SQL_CREATE_ENTRIES);
    }
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int
newVersion) {
        // This database is only a cache for online data, so its
upgrade policy is
        // to simply to discard the data and start over
        db.execSQL(SQL_DELETE_ENTRIES);
        onCreate(db);
```

```
    }
    public void onDowngrade(SQLiteDatabase db, int oldVersion, int
newVersion) {
        onUpgrade(db, oldVersion, newVersion);
    }
}
```

To access your database, instantiate your subclass of `SQLiteOpenHelper`:

```
FeedReaderDbHelper mDbHelper = new FeedReaderDbHelper(getContext());
```

Put Information into a Database

Insert data into the database by passing a `ContentValues` object to the `insert()` method:

```
// Gets the data repository in write mode
SQLiteDatabase db = mDbHelper.getWritableDatabase();

// Create a new map of values, where column names are the keys
ContentValues values = new ContentValues();
values.put(FeedEntry.COLUMN_NAME_ENTRY_ID, id);
values.put(FeedEntry.COLUMN_NAME_TITLE, title);
values.put(FeedEntry.COLUMN_NAME_CONTENT, content);

// Insert the new row, returning the primary key value of the new
row
long newRowId;
newRowId = db.insert(
         FeedEntry.TABLE_NAME,
         FeedEntry.COLUMN_NAME_NULLABLE,
         values);
```

The first argument for `insert()` is simply the table name. The second argument provides the name of a column in which the framework can insert NULL in the event that the `ContentValues` is empty (if you instead set this to `"null"`, then the framework will not insert a row when there are no values).

Read Information from a Database

To read from a database, use the `query()` method, passing it your selection criteria and desired columns. The method combines elements of `insert()` and `update()`, except the column list defines the data you want to fetch, rather than the data to insert. The results of the query are returned to you in a `Cursor` object.

```
SQLiteDatabase db = mDbHelper.getReadableDatabase();

// Define a projection that specifies which columns from the
database
// you will actually use after this query.
String[] projection = {
    FeedEntry._ID,
    FeedEntry.COLUMN_NAME_TITLE,
    FeedEntry.COLUMN_NAME_UPDATED,
    ...
    };

// How you want the results sorted in the resulting Cursor
String sortOrder =
    FeedEntry.COLUMN_NAME_UPDATED + " DESC";

Cursor c = db.query(
    FeedEntry.TABLE_NAME,  // The table to query
    projection,                               // The columns to
return
    selection,                                // The columns for the
WHERE clause
    selectionArgs,                            // The values for the
WHERE clause
    null,                                     // don't group the
rows
    null,                                     // don't filter by row
groups
    sortOrder                                 // The sort order
    );
```

To look at a row in the cursor, use one of the `Cursor` move methods, which you must always call before you begin reading values. Generally, you should start by calling `moveToFirst()`, which places the "read position" on the first entry in the results. For each row, you can read a column's value by calling one of the `Cursor` get methods, such as `getString()` or `getLong()`. For each of the get methods, you must pass the index position of the column you desire, which you can get by calling `getColumnIndex()` or `getColumnIndexOrThrow()`. For example:

```
cursor.moveToFirst();
long itemId = cursor.getLong(
    cursor.getColumnIndexOrThrow(FeedEntry._ID)
);
```

Delete Information from a Database

To delete rows from a table, you need to provide selection criteria that identify the rows. The database API provides a mechanism for creating selection criteria that protects against SQL injection. The mechanism

divides the selection specification into a selection clause and selection arguments. The clause defines the columns to look at, and also allows you to combine column tests. The arguments are values to test against that are bound into the clause. Because the result isn't handled the same as a regular SQL statement, it is immune to SQL injection.

```
// Define 'where' part of query.
String selection = FeedEntry.COLUMN_NAME_ENTRY_ID + " LIKE ?";
// Specify arguments in placeholder order.
String[] selectionArgs = { String.valueOf(rowId) };
// Issue SQL statement.
db.delete(table_name, selection, selectionArgs);
```

Update a Database

When you need to modify a subset of your database values, use the `update()` method.

Updating the table combines the content values syntax of `insert()` with the `where` syntax of `delete()`.

```
SQLiteDatabase db = mDbHelper.getReadableDatabase();

// New value for one column
ContentValues values = new ContentValues();
values.put(FeedEntry.COLUMN_NAME_TITLE, title);

// Which row to update, based on the ID
String selection = FeedEntry.COLUMN_NAME_ENTRY_ID + " LIKE ?";
String[] selectionArgs = { String.valueOf(rowId) };

int count = db.update(
    FeedReaderDbHelper.FeedEntry.TABLE_NAME,
    values,
    selection,
    selectionArgs);
```

Next Get started Next class

Interacting with Other Apps

Previous Next Get started

Dependencies and prerequisites

- Basic understanding of the Activity lifecycle (see Managing the Activity Lifecycle)

You should also read

- Sharing Simple Data
- Sharing Files
- Integrating Application with Intents (blog post)
- Intents and Intent Filters

An Android app typically has several activities. Each activity displays a user interface that allows the user to perform a specific task (such as view a map or take a photo). To take the user from one activity to another, your app must use an `Intent` to define your app's "intent" to do something. When you pass an `Intent` to the system with a method such as `startActivity()`, the system uses the `Intent` to identify and start the appropriate app component. Using intents even allows your app to start an activity that is contained in a separate app.

An `Intent` can be *explicit* in order to start a specific component (a specific `Activity` instance) or *implicit* in order to start any component that can handle the intended action (such as "capture a photo").

This class shows you how to use an `Intent` to perform some basic interactions with other apps, such as start another app, receive a result from that app, and make your app able to respond to intents from other apps.

Lessons

Sending the User to Another App

Shows how you can create implicit intents to launch other apps that can perform an action.

Getting a Result from an Activity

Shows how to start another activity and receive a result from the activity.

Allowing Other Apps to Start Your Activity

Shows how to make activities in your app open for use by other apps by defining intent filters that declare the implicit intents your app accepts.

Next Get started Next class

Sending the User to Another App

Previous Next Get started

This lesson teaches you to

1. Build an Implicit Intent
2. Verify There is an App to Receive the Intent
3. Start an Activity with the Intent
4. Show an App Chooser

You should also read

- Sharing Simple Data

One of Android's most important features is an app's ability to send the user to another app based on an "action" it would like to perform. For example, if your app has the address of a business that you'd like to show on a map, you don't have to build an activity in your app that shows a map. Instead, you can create a request to view the address using an `Intent`. The Android system then starts an app that's able to show the address on a map.

As explained in the first class, Building Your First App, you must use intents to navigate between activities in your own app. You generally do so with an *explicit intent*, which defines the exact class name of the component you want to start. However, when you want to have a separate app perform an action, such as "view a map," you must use an *implicit intent*.

This lesson shows you how to create an implicit intent for a particular action, and how to use it to start an activity that performs the action in another app.

Build an Implicit Intent

Implicit intents do not declare the class name of the component to start, but instead declare an action to perform. The action specifies the thing you want to do, such as *view*, *edit*, *send*, or *get* something. Intents often also include data associated with the action, such as the address you want to view, or the email message you want to send. Depending on the intent you want to create, the data might be a `Uri`, one of several other data types, or the intent might not need data at all.

If your data is a `Uri`, there's a simple `Intent()` constructor you can use define the action and data.

For example, here's how to create an intent to initiate a phone call using the `Uri` data to specify the telephone number:

```
Uri number = Uri.parse("tel:5551234");
Intent callIntent = new Intent(Intent.ACTION_DIAL, number);
```

When your app invokes this intent by calling `startActivity()`, the Phone app initiates a call to the given phone number.

Here are a couple other intents and their action and `Uri` data pairs:

- View a map:

  ```
  // Map point based on address
  Uri location = Uri.parse("geo:0,0?
  q=1600+Amphitheatre+Parkway,+Mountain+View,+California");
  // Or map point based on latitude/longitude
  // Uri location = Uri.parse("geo:37.422219,-122.08364?z=14"); //
  z param is zoom level
  Intent mapIntent = new Intent(Intent.ACTION_VIEW, location);
  ```

- View a web page:

  ```
  Uri webpage = Uri.parse("http://www.android.com");
  Intent webIntent = new Intent(Intent.ACTION_VIEW, webpage);
  ```

Other kinds of implicit intents require "extra" data that provide different data types, such as a string. You can add one or more pieces of extra data using the various `putExtra()` methods.

By default, the system determines the appropriate MIME type required by an intent based on the `Uri` data that's included. If you don't include a `Uri` in the intent, you should usually use `setType()` to specify the type of data associated with the intent. Setting the MIME type further specifies which kinds of activities should receive the intent.

Here are some more intents that add extra data to specify the desired action:

- Send an email with an attachment:

```
Intent emailIntent = new Intent(Intent.ACTION_SEND);
// The intent does not have a URI, so declare the "text/plain"
MIME type
emailIntent.setType(HTTP.PLAIN_TEXT_TYPE);
emailIntent.putExtra(Intent.EXTRA_EMAIL, new String[]
{"jon@example.com"}); // recipients
emailIntent.putExtra(Intent.EXTRA_SUBJECT, "Email subject");
emailIntent.putExtra(Intent.EXTRA_TEXT, "Email message text");
emailIntent.putExtra(Intent.EXTRA_STREAM,
Uri.parse("content://path/to/email/attachment"));
// You can also attach multiple items by passing an ArrayList
of Uris
```

- Create a calendar event:

```
Intent calendarIntent = new Intent(Intent.ACTION_INSERT,
Events.CONTENT_URI);
Calendar beginTime = Calendar.getInstance().set(2012, 0, 19, 7,
30);
Calendar endTime = Calendar.getInstance().set(2012, 0, 19, 10,
30);
calendarIntent.putExtra(CalendarContract.EXTRA_EVENT_BEGIN_TIME,
beginTime.getTimeInMillis());
calendarIntent.putExtra(CalendarContract.EXTRA_EVENT_END_TIME,
endTime.getTimeInMillis());
calendarIntent.putExtra(Events.TITLE, "Ninja class");
calendarIntent.putExtra(Events.EVENT_LOCATION, "Secret dojo");
```

  Note: This intent for a calendar event is supported only with API level 14 and higher.

Note: It's important that you define your `Intent` to be as specific as possible. For example, if you want to display an image using the `ACTION_VIEW` intent, you should specify a MIME type of `image/*`. This prevents apps that can "view" other types of data (like a map app) from being triggered by the intent.

Verify There is an App to Receive the Intent

Although the Android platform guarantees that certain intents will resolve to one of the built-in apps (such as the Phone, Email, or Calendar app), you should always include a verification step before invoking an intent.

Caution: If you invoke an intent and there is no app available on the device that can handle the intent, your app will crash.

To verify there is an activity available that can respond to the intent, call `queryIntentActivities()` to get a list of activities capable of handling your `Intent`. If the returned `List` is not empty, you can safely use the intent. For example:

```
PackageManager packageManager = getPackageManager();
List activities = packageManager.queryIntentActivities(intent,
        PackageManager.MATCH_DEFAULT_ONLY);
boolean isIntentSafe = activities.size() > 0;
```

If `isIntentSafe` is `true`, then at least one app will respond to the intent. If it is `false`, then there aren't any apps to handle the intent.

Note: You should perform this check when your activity first starts in case you need to disable the feature that uses the intent before the user attempts to use it. If you know of a specific app that can handle the intent, you can also provide a link for the user to download the app (see how to link to your product on Google Play).

Start an Activity with the Intent

Figure 1. Example of the selection dialog

that appears when more than one app can handle an intent.

Once you have created your `Intent` and set the extra info, call `startActivity()` to send it to the system. If the system identifies more than one activity that can handle the intent, it displays a dialog for the user to select which app to use, as shown in figure 1. If there is only one activity that handles the intent, the system immediately starts it.

```
startActivity(intent);
```

Here's a complete example that shows how to create an intent to view a map, verify that an app exists to handle the intent, then start it:

```
// Build the intent
Uri location = Uri.parse("geo:0,0?
q=1600+Amphitheatre+Parkway,+Mountain+View,+California");
Intent mapIntent = new Intent(Intent.ACTION_VIEW, location);

// Verify it resolves
PackageManager packageManager = getPackageManager();
List<ResolveInfo> activities =
packageManager.queryIntentActivities(mapIntent, 0);
boolean isIntentSafe = activities.size() > 0;

// Start an activity if it's safe
if (isIntentSafe) {
    startActivity(mapIntent);
}
```

Show an App Chooser

Figure 2. A chooser dialog.

Notice that when you start an activity by passing your `Intent` to `startActivity()` and there is more than one app that responds to the intent, the user can select which app to use by default (by selecting a checkbox at the bottom of the dialog; see figure 1). This is nice when performing an action for which the user generally wants to use the same app every time, such as when opening a web page (users likely use just one web browser) or taking a photo (users likely prefer one camera).

However, if the action to be performed could be handled by multiple apps and the user might prefer a different app each time—such as a "share" action, for which users might have several apps through which they might share an item—you should explicitly show a chooser dialog as shown in figure 2. The chooser dialog forces the user to select which app to use for the action every time (the user cannot select a default app for the action).

To show the chooser, create an `Intent` using `createChooser()` and pass it to `startActivity()`. For example:

```
Intent intent = new Intent(Intent.ACTION_SEND);
...

// Always use string resources for UI text.
```

```
// This says something like "Share this photo with"
String title = getResources().getString(R.string.chooser_title);
// Create intent to show chooser
Intent chooser = Intent.createChooser(intent, title);

// Verify the intent will resolve to at least one activity
if (intent.resolveActivity(getPackageManager()) != null) {
    startActivity(chooser);
}
```

This displays a dialog with a list of apps that respond to the intent passed to the `createChooser()` method and uses the supplied text as the dialog title.

Next Get started Next class

Getting a Result from an Activity

Previous Next Get started

This lesson teaches you to

1. Start the Activity
2. Receive the Result

You should also read

- Sharing Simple Data
- Sharing Files

Starting another activity doesn't have to be one-way. You can also start another activity and receive a result back. To receive a result, call `startActivityForResult()` (instead of `startActivity()`).

For example, your app can start a camera app and receive the captured photo as a result. Or, you might start the People app in order for the user to select a contact and you'll receive the contact details as a result.

Of course, the activity that responds must be designed to return a result. When it does, it sends the result as another `Intent` object. Your activity receives it in the `onActivityResult()` callback.

Note: You can use explicit or implicit intents when you call `startActivityForResult()`. When starting one of your own activities to receive a result, you should use an explicit intent to ensure that you receive the expected result.

Start the Activity

There's nothing special about the `Intent` object you use when starting an activity for a result, but you do need to pass an additional integer argument to the `startActivityForResult()` method.

The integer argument is a "request code" that identifies your request. When you receive the result `Intent`, the callback provides the same request code so that your app can properly identify the result and determine how to handle it.

For example, here's how to start an activity that allows the user to pick a contact:

```
static final int PICK_CONTACT_REQUEST = 1;  // The request code
...
private void pickContact() {
    Intent pickContactIntent = new Intent(Intent.ACTION_PICK,
Uri.parse("content://contacts"));
    pickContactIntent.setType(Phone.CONTENT_TYPE); // Show user only
contacts w/ phone numbers
    startActivityForResult(pickContactIntent, PICK_CONTACT_REQUEST);
}
```

Receive the Result

When the user is done with the subsequent activity and returns, the system calls your activity's `onActivityResult()` method. This method includes three arguments:

- The request code you passed to `startActivityForResult()`.
- A result code specified by the second activity. This is either `RESULT_OK` if the operation was successful or `RESULT_CANCELED` if the user backed out or the operation failed for some reason.
- An `Intent` that carries the result data.

For example, here's how you can handle the result for the "pick a contact" intent:

```
@Override
protected void onActivityResult(int requestCode, int resultCode,
Intent data) {
    // Check which request we're responding to
    if (requestCode == PICK_CONTACT_REQUEST) {
        // Make sure the request was successful
        if (resultCode == RESULT_OK) {
```

```
            // The user picked a contact.
            // The Intent's data Uri identifies which contact was
selected.

            // Do something with the contact here (bigger example
below)
        }
    }
}
```

In this example, the result `Intent` returned by Android's Contacts or People app provides a content `Uri` that identifies the contact the user selected.

In order to successfully handle the result, you must understand what the format of the result `Intent` will be. Doing so is easy when the activity returning a result is one of your own activities. Apps included with the Android platform offer their own APIs that you can count on for specific result data. For instance, the People app (Contacts app on some older versions) always returns a result with the content URI that identifies the selected contact, and the Camera app returns a `Bitmap` in the `"data"` extra (see the class about Capturing Photos).

Bonus: Read the contact data

The code above showing how to get a result from the People app doesn't go into details about how to actually read the data from the result, because it requires more advanced discussion about content providers. However, if you're curious, here's some more code that shows how to query the result data to get the phone number from the selected contact:

```
@Override
protected void onActivityResult(int requestCode, int resultCode,
Intent data) {
    // Check which request it is that we're responding to
    if (requestCode == PICK_CONTACT_REQUEST) {
        // Make sure the request was successful
        if (resultCode == RESULT_OK) {
            // Get the URI that points to the selected contact
            Uri contactUri = data.getData();
            // We only need the NUMBER column, because there will be
only one row in the result
            String[] projection = {Phone.NUMBER};

            // Perform the query on the contact to get the NUMBER
column
            // We don't need a selection or sort order (there's only
one result for the given URI)
```

```
            // CAUTION: The query() method should be called from a
separate thread to avoid blocking
            // your app's UI thread. (For simplicity of the sample,
this code doesn't do that.)
            // Consider using CursorLoader to perform the query.
            Cursor cursor = getContentResolver()
                    .query(contactUri, projection, null, null, null);
            cursor.moveToFirst();

            // Retrieve the phone number from the NUMBER column
            int column = cursor.getColumnIndex(Phone.NUMBER);
            String number = cursor.getString(column);

            // Do something with the phone number...
        }
    }
}
```

Note: Before Android 2.3 (API level 9), performing a query on the `Contacts Provider` (like the one shown above) requires that your app declare the `READ_CONTACTS` permission (see Security and Permissions). However, beginning with Android 2.3, the Contacts/People app grants your app a temporary permission to read from the Contacts Provider when it returns you a result. The temporary permission applies only to the specific contact requested, so you cannot query a contact other than the one specified by the intent's `Uri`, unless you do declare the `READ_CONTACTS` permission.

Next Get started Next class

Allowing Other Apps to Start Your Activity

Previous Next Get started

This lesson teaches you to

1. Add an Intent Filter
2. Handle the Intent in Your Activity
3. Return a Result

You should also read

- Sharing Simple Data
- Sharing Files

The previous two lessons focused on one side of the story: starting another app's activity from your app. But if your app can perform an action that might be useful to another app, your app should be prepared to respond to action requests from other apps. For instance, if you build a social app that can share messages or photos with the user's friends, it's in your best interest to support the `ACTION_SEND` intent so users can initiate a "share" action from another app and launch your app to perform the action.

To allow other apps to start your activity, you need to add an `<intent-filter>` element in your manifest file for the corresponding `<activity>` element.

When your app is installed on a device, the system identifies your intent filters and adds the information to an internal catalog of intents supported by all installed apps. When an app calls `startActivity()` or `startActivityForResult()`, with an implicit intent, the system finds which

activity (or activities) can respond to the intent.

Add an Intent Filter

In order to properly define which intents your activity can handle, each intent filter you add should be as specific as possible in terms of the type of action and data the activity accepts.

The system may send a given `Intent` to an activity if that activity has an intent filter fulfills the following criteria of the `Intent` object:

Action
: A string naming the action to perform. Usually one of the platform-defined values such as `ACTION_SEND` or `ACTION_VIEW`.

  Specify this in your intent filter with the `<action>` element. The value you specify in this element must be the full string name for the action, instead of the API constant (see the examples below).

Data
: A description of the data associated with the intent.

  Specify this in your intent filter with the `<data>` element. Using one or more attributes in this element, you can specify just the MIME type, just a URI prefix, just a URI scheme, or a combination of these and others that indicate the data type accepted.

  Note: If you don't need to declare specifics about the data `Uri` (such as when your activity handles to other kind of "extra" data, instead of a URI), you should specify only the `android:mimeType` attribute to declare the type of data your activity handles, such as `text/plain` or `image/jpeg`.

Category
: Provides an additional way to characterize the activity handling the intent, usually related to the user gesture or location from which it's started. There are several different categories supported by the system, but most are rarely used. However, all implicit intents are

defined with CATEGORY_DEFAULT by default.

Specify this in your intent filter with the <category> element.

In your intent filter, you can declare which criteria your activity accepts by declaring each of them with corresponding XML elements nested in the <intent-filter> element.

For example, here's an activity with an intent filter that handles the ACTION_SEND intent when the data type is either text or an image:

```
<activity android:name="ShareActivity">
    <intent-filter>
        <action android:name="android.intent.action.SEND"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <data android:mimeType="text/plain"/>
        <data android:mimeType="image/*"/>
    </intent-filter>
</activity>
```

Each incoming intent specifies only one action and one data type, but it's OK to declare multiple instances of the <action>, <category>, and <data> elements in each <intent-filter>.

If any two pairs of action and data are mutually exclusive in their behaviors, you should create separate intent filters to specify which actions are acceptable when paired with which data types.

For example, suppose your activity handles both text and images for both the ACTION_SEND and ACTION_SENDTO intents. In this case, you must define two separate intent filters for the two actions because a ACTION_SENDTO intent must use the data Uri to specify the recipient's address using the `send` or `sendto` URI scheme. For example:

```
<activity android:name="ShareActivity">
    <!-- filter for sending text; accepts SENDTO action with sms URI
schemes -->
    <intent-filter>
        <action android:name="android.intent.action.SENDTO"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <data android:scheme="sms" />
        <data android:scheme="smsto" />
    </intent-filter>
    <!-- filter for sending text or images; accepts SEND action and
text or image data -->
    <intent-filter>
        <action android:name="android.intent.action.SEND"/>
```

```
        <category android:name="android.intent.category.DEFAULT"/>
        <data android:mimeType="image/*"/>
        <data android:mimeType="text/plain"/>
    </intent-filter>
</activity>
```

Note: In order to receive implicit intents, you must include the `CATEGORY_DEFAULT` category in the intent filter. The methods `startActivity()` and `startActivityForResult()` treat all intents as if they declared the `CATEGORY_DEFAULT` category. If you do not declare it in your intent filter, no implicit intents will resolve to your activity.

For more information about sending and receiving `ACTION_SEND` intents that perform social sharing behaviors, see the lesson about Receiving Simple Data from Other Apps.

Handle the Intent in Your Activity

In order to decide what action to take in your activity, you can read the `Intent` that was used to start it.

As your activity starts, call `getIntent()` to retrieve the `Intent` that started the activity. You can do so at any time during the lifecycle of the activity, but you should generally do so during early callbacks such as `onCreate()` or `onStart()`.

For example:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setContentView(R.layout.main);

    // Get the intent that started this activity
    Intent intent = getIntent();
    Uri data = intent.getData();

    // Figure out what to do based on the intent type
    if (intent.getType().indexOf("image/") != -1) {
        // Handle intents with image data ...
    } else if (intent.getType().equals("text/plain")) {
        // Handle intents with text ...
    }
}
```

Return a Result

If you want to return a result to the activity that invoked yours, simply call `setResult()` to specify the result code and result `Intent`. When your operation is done and the user should return to the original activity, call `finish()` to close (and destroy) your activity. For example:

```
// Create intent to deliver some kind of result data
Intent result = new Intent("com.example.RESULT_ACTION",
Uri.parse("content://result_uri");
setResult(Activity.RESULT_OK, result);
finish();
```

You must always specify a result code with the result. Generally, it's either `RESULT_OK` or `RESULT_CANCELED`. You can then provide additional data with an `Intent`, as necessary.

Note: The result is set to `RESULT_CANCELED` by default. So, if the user presses the *Back* button before completing the action and before you set the result, the original activity receives the "canceled" result.

If you simply need to return an integer that indicates one of several result options, you can set the result code to any value higher than 0. If you use the result code to deliver an integer and you have no need to include the `Intent`, you can call `setResult()` and pass only a result code. For example:

```
setResult(RESULT_COLOR_RED);
finish();
```

In this case, there might be only a handful of possible results, so the result code is a locally defined integer (greater than 0). This works well when you're returning a result to an activity in your own app, because the activity that receives the result can reference the public constant to determine the value of the result code.

Note: There's no need to check whether your activity was started with `startActivity()` or `startActivityForResult()`. Simply call `setResult()` if the intent that started your activity might expect a result. If the originating activity had called `startActivityForResult()`, then the system delivers it the result you supply to `setResult()`; otherwise, the result is

ignored.

Next Get started Next class

Building Apps with Content Sharing

These classes teach you how to create apps that share data between apps and devices.

Sharing Simple Data

Previous Next Get started

Dependencies and prerequisites

- Android 1.0 or higher (greater requirements where noted)
- Experience with Intents and Intent Filters

One of the great things about Android applications is their ability to communicate and integrate with each other. Why reinvent functionality that isn't core to your application when it already exists in another application?

This class covers some common ways you can send and receive simple data between applications using `Intent` APIs and the `ActionProvider` object.

Lessons

Sending Simple Data to Other Apps
Learn how to set up your application to be able to send text and binary data to other applications with intents.

Receiving Simple Data from Other Apps
Learn how to set up your application to receive text and binary data from intents.

Adding an Easy Share Action
Learn how to add a "share" action item to your action bar.

Next Get started Next class

Sending Simple Data to Other Apps

Previous Next Get started

This lesson teaches you to

1. Send Text Content
2. Send Binary Content
3. Send Multiple Pieces of Content

You should also read

- Intents and Intent Filters

When you construct an intent, you must specify the action you want the intent to "trigger." Android defines several actions, including `ACTION_SEND` which, as you can probably guess, indicates that the intent is sending data from one activity to another, even across process boundaries. To send data to another activity, all you need to do is specify the data and its type, the system will identify compatible receiving activities and display them to the user (if there are multiple options) or immediately start the activity (if there is only one option). Similarly, you can advertise the data types that your activities support receiving from other applications by specifying them in your manifest.

Sending and receiving data between applications with intents is most commonly used for social sharing of content. Intents allow users to share information quickly and easily, using their favorite applications.

Note: The best way to add a share action item to an `ActionBar` is to use `ShareActionProvider`, which became available in API level 14. `ShareActionProvider` is discussed in the lesson about Adding an Easy

Share Action.

Send Text Content

Figure 1. Screenshot of `ACTION_SEND` intent chooser on a handset.

The most straightforward and common use of the `ACTION_SEND` action is sending text content from one activity to another. For example, the built-in Browser app can share the URL of the currently-displayed page as text with any application. This is useful for sharing an article or website with friends via email or social networking. Here is the code to implement this type of sharing:

```
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, "This is my text to send.");
sendIntent.setType("text/plain");
startActivity(sendIntent);
```

If there's an installed application with a filter that matches `ACTION_SEND` and MIME type text/plain, the Android system will run it; if more than

one application matches, the system displays a disambiguation dialog (a "chooser") that allows the user to choose an app.

However, if you call `Intent.createChooser()`, passing it your `Intent` object, it returns a version of your intent that will **always display the chooser**. This has some advantages:

- Even if the user has previously selected a default action for this intent, the chooser will still be displayed.
- If no applications match, Android displays a system message.
- You can specify a title for the chooser dialog.

Here's the updated code:

```
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, "This is my text to send.");
sendIntent.setType("text/plain");
startActivity(Intent.createChooser(sendIntent,
getResources().getText(R.string.send_to));
```

The resulting dialog is shown in figure 1.

Optionally, you can set some standard extras for the intent: `EXTRA_EMAIL`, `EXTRA_CC`, `EXTRA_BCC`, `EXTRA_SUBJECT`. If the receiving application is not designed to use them, it simply ignores them.

Note: Some e-mail applications, such as Gmail, expect a `String[]` for extras like `EXTRA_EMAIL` and `EXTRA_CC`, use `putExtra(String, String[])` to add these to your intent.

Send Binary Content

Binary data is shared using the `ACTION_SEND` action combined with setting the appropriate MIME type and placing the URI to the data in an extra named `EXTRA_STREAM`. This is commonly used to share an image but can be used to share any type of binary content:

```
Intent shareIntent = new Intent();
shareIntent.setAction(Intent.ACTION_SEND);
shareIntent.putExtra(Intent.EXTRA_STREAM, uriToImage);
shareIntent.setType("image/jpeg");
```

```
startActivity(Intent.createChooser(shareIntent,
getResources().getText(R.string.send_to)));
```

Note the following:

- You can use a MIME type of `"*/*"`, but this will only match activities that are able to handle generic data streams.
- The receiving application needs permission to access the data the `Uri` points to. The recommended ways to do this are:
  - Store the data in your own `ContentProvider`, making sure that other apps have the correct permission to access your provider. The preferred mechanism for providing access is to use per-URI permissions which are temporary and only grant access to the receiving application. An easy way to create a `ContentProvider` like this is to use the `FileProvider` helper class.
  - Use the system `MediaStore`. The `MediaStore` is primarily aimed at video, audio and image MIME types, however beginning with Android 3.0 (API level 11) it can also store non-media types (see `MediaStore.Files` for more info). Files can be inserted into the `MediaStore` using `scanFile()` after which a `content://` style `Uri` suitable for sharing is passed to the provided `onScanCompleted()` callback. Note that once added to the system `MediaStore` the content is accessible to any app on the device.

Send Multiple Pieces of Content

To share multiple pieces of content, use the `ACTION_SEND_MULTIPLE` action together with a list of URIs pointing to the content. The MIME type varies according to the mix of content you're sharing. For example, if you share 3 JPEG images, the type is still `"image/jpeg"`. For a mixture of image types, it should be `"image/*"` to match an activity that handles any type of image. You should only use `"*/*"` if you're sharing out a wide variety of types. As previously stated, it's up to the receiving application to parse and process your data. Here's an example:

```
ArrayList<Uri> imageUris = new ArrayList<Uri>();
imageUris.add(imageUri1); // Add your image URIs here
imageUris.add(imageUri2);

Intent shareIntent = new Intent();
```

```
shareIntent.setAction(Intent.ACTION_SEND_MULTIPLE);
shareIntent.putParcelableArrayListExtra(Intent.EXTRA_STREAM,
imageUris);
shareIntent.setType("image/*");
startActivity(Intent.createChooser(shareIntent, "Share images
to.."));
```

As before, make sure the provided `URIs` point to data that a receiving application can access.

Next Get started Next class

Receiving Simple Data from Other Apps

Previous Next Get started

This lesson teaches you to

1. Update Your Manifest
2. Handle the Incoming Content

You should also read

- Intents and Intent Filters

Just as your application can send data to other applications, so too can it easily receive data from applications. Think about how users interact with your application, and what data types you want to receive from other applications. For example, a social networking application would likely be interested in receiving text content, like an interesting web URL, from another app. The Google+ Android application accepts both text *and* single or multiple images. With this app, a user can easily start a new Google+ post with photos from the Android Gallery app.

Update Your Manifest

Intent filters inform the system what intents an application component is willing to accept. Similar to how you constructed an intent with action `ACTION_SEND` in the Sending Simple Data to Other Apps lesson, you create intent filters in order to be able to receive intents with this action. You define an intent filter in your manifest, using the `<intent-filter>` element. For example, if your application handles receiving text content, a single image of any type, or multiple images of any type, your manifest

would look like:

```
<activity android:name=".ui.MyActivity" >
    <intent-filter>
        <action android:name="android.intent.action.SEND" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="image/*" />
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.SEND" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="text/plain" />
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.SEND_MULTIPLE" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="image/*" />
    </intent-filter>
</activity>
```

Note: For more information on intent filters and intent resolution please read Intents and Intent Filters

When another application tries to share any of these things by constructing an intent and passing it to `startActivity()`, your application will be listed as an option in the intent chooser. If the user selects your application, the corresponding activity (`.ui.MyActivity` in the example above) will be started. It is then up to you to handle the content appropriately within your code and UI.

Handle the Incoming Content

To handle the content delivered by an `Intent`, start by calling `getIntent()` to get `Intent` object. Once you have the object, you can examine its contents to determine what to do next. Keep in mind that if this activity can be started from other parts of the system, such as the launcher, then you will need to take this into consideration when examining the intent.

```
void onCreate (Bundle savedInstanceState) {
    ...
    // Get intent, action and MIME type
    Intent intent = getIntent();
    String action = intent.getAction();
    String type = intent.getType();

    if (Intent.ACTION_SEND.equals(action) && type != null) {
        if ("text/plain".equals(type)) {
            handleSendText(intent); // Handle text being sent
```

```
        } else if (type.startsWith("image/")) {
            handleSendImage(intent); // Handle single image being
sent
        }
    } else if (Intent.ACTION_SEND_MULTIPLE.equals(action) && type !=
null) {
        if (type.startsWith("image/")) {
            handleSendMultipleImages(intent); // Handle multiple
images being sent
        }
    } else {
        // Handle other intents, such as being started from the home
screen
    }
    ...
}

void handleSendText(Intent intent) {
    String sharedText = intent.getStringExtra(Intent.EXTRA_TEXT);
    if (sharedText != null) {
        // Update UI to reflect text being shared
    }
}

void handleSendImage(Intent intent) {
    Uri imageUri = (Uri)
intent.getParcelableExtra(Intent.EXTRA_STREAM);
    if (imageUri != null) {
        // Update UI to reflect image being shared
    }
}

void handleSendMultipleImages(Intent intent) {
    ArrayList<Uri> imageUris =
intent.getParcelableArrayListExtra(Intent.EXTRA_STREAM);
    if (imageUris != null) {
        // Update UI to reflect multiple images being shared
    }
}
```

Caution: Take extra care to check the incoming data, you never know what some other application may send you. For example, the wrong MIME type might be set, or the image being sent might be extremely large. Also, remember to process binary data in a separate thread rather than the main ("UI") thread.

Updating the UI can be as simple as populating an `EditText`, or it can be more complicated like applying an interesting photo filter to an image. It's really specific to your application what happens next.

Next Get started Next class

Adding an Easy Share Action

Previous Next Get started

This lesson teaches you to

1. Update Menu Declarations
2. Set the Share Intent

You should also read

- Action Bar

Implementing an effective and user friendly share action in your `ActionBar` is made even easier with the introduction of `ActionProvider` in Android 4.0 (API Level 14). An `ActionProvider`, once attached to a menu item in the action bar, handles both the appearance and behavior of that item. In the case of `ShareActionProvider`, you provide a share intent and it does the rest.

Note: `ShareActionProvider` is available starting with API Level 14 and higher.

Figure 1. The `ShareActionProvider` in the Gallery app.

Update Menu Declarations

To get started with `ShareActionProviders`, define the `android:actionProviderClass` attribute for the corresponding `<item>` in your menu resource file:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
            android:id="@+id/menu_item_share"
            android:showAsAction="ifRoom"
            android:title="Share"
            android:actionProviderClass=
                "android.widget.ShareActionProvider" />
    ...
</menu>
```

This delegates responsibility for the item's appearance and function to `ShareActionProvider`. However, you will need to tell the provider what

you would like to share.

Set the Share Intent

In order for `ShareActionProvider` to function, you must provide it a share intent. This share intent should be the same as described in the Sending Simple Data to Other Apps lesson, with action `ACTION_SEND` and additional data set via extras like `EXTRA_TEXT` and `EXTRA_STREAM`. To assign a share intent, first find the corresponding `MenuItem` while inflating your menu resource in your `Activity` or `Fragment`. Next, call `MenuItem.getActionProvider()` to retrieve an instance of `ShareActionProvider`. Use `setShareIntent()` to update the share intent associated with that action item. Here's an example:

```
private ShareActionProvider mShareActionProvider;
...

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate menu resource file.
    getMenuInflater().inflate(R.menu.share_menu, menu);

    // Locate MenuItem with ShareActionProvider
    MenuItem item = menu.findItem(R.id.menu_item_share);

    // Fetch and store ShareActionProvider
    mShareActionProvider = (ShareActionProvider)
item.getActionProvider();

    // Return true to display menu
    return true;
}

// Call to update the share intent
private void setShareIntent(Intent shareIntent) {
    if (mShareActionProvider != null) {
        mShareActionProvider.setShareIntent(shareIntent);
    }
}
```

You may only need to set the share intent once during the creation of your menus, or you may want to set it and then update it as the UI changes. For example, when you view photos full screen in the Gallery app, the sharing intent changes as you flip between photos.

For further discussion about the `ShareActionProvider` object, see the Action Bar guide.

Next Get started Next class

Sharing Files

Previous Next Get started

Dependencies and prerequisites

- Android 1.6 (API Level 4) or higher
- Familiarity with file operations such as opening, reading, and writing files

You should also read

- Storage Options
- Saving Files
- Sharing Simple Data

Apps often have a need to offer one or more of their files to another app. For example, an image gallery may want to offer files to image editors, or a file management app may want to allow users to copy and paste files between areas in external storage. One way a sending app can share a file is to respond to a request from the receiving app.

In all cases, the only secure way to offer a file from your app to another app is to send the receiving app the file's content URI and grant temporary access permissions to that URI. Content URIs with temporary URI access permissions are secure because they apply only to the app that receives the URI, and they expire automatically. The Android `FileProvider` component provides the method `getUriForFile()` for generating a file's content URI.

If you want to share small amounts of text or numeric data between apps, you should send an `Intent` that contains the data. To learn how to send simple data with an `Intent`, see the training class Sharing Simple

Data.

This class explains how to securely share files from your app to another app using content URIs generated by the Android `FileProvider` component and temporary permissions that you grant to the receiving app for the content URI.

Lessons

Setting Up File Sharing
: Learn how to set up your app to share files.

Sharing a File
: Learn how to offer a file to another app by generating a content URI for the file, granting access permissions to the URI, and sending the URI to the app.

Requesting a Shared File
: Learn how to request a file shared by another app, receive the content URI for the file, and use the content URI to open the file.

Retrieving File Information
: Learn how an app can use a content URI generated by a `FileProvider` to retrieve file information including MIME type and file size.

Next Get started Next class

Setting Up File Sharing

Previous Next Get started

This lesson teaches you to

1. Specify the FileProvider
2. Specify Sharable Directories

You should also read

- Storage Options
- Saving Files

To securely offer a file from your app to another app, you need to configure your app to offer a secure handle to the file, in the form of a content URI. The Android `FileProvider` component generates content URIs for files, based on specifications you provide in XML. This lesson shows you how to add the default implementation of `FileProvider` to your app, and how to specify the files you want to offer to other apps.

Note: The `FileProvider` class is part of the v4 Support Library. For information about including this library in your application, see Support Library Setup.

Specify the FileProvider

Defining a `FileProvider` for your app requires an entry in your manifest. This entry specifies the authority to use in generating content URIs, as well as the name of an XML file that specifies the directories your app can share.

The following snippet shows you how to add to your manifest the

`<provider>` element that specifies the `FileProvider` class, the authority, and the XML file name:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.myapp">
    <application
        ...>
        <provider
            android:name="android.support.v4.content.FileProvider"
            android:authorities="com.example.myapp.fileprovider"
            android:grantUriPermissions="true"
            android:exported="false">
            <meta-data
                android:name="android.support.FILE_PROVIDER_PATHS"
                android:resource="@xml/filepaths" />
        </provider>
        ...
    </application>
</manifest>
```

In this example, the `android:authorities` attribute specifies the URI authority that you want to use for content URIs generated by the `FileProvider`. In the example, the authority is `com.example.myapp.fileprovider`. For your own app, specify an authority consisting of the app's `android:package` value with the string "fileprovider" appended to it. To learn more about the authority value, see the topic Content URIs and the documentation for the `android:authorities` attribute.

The `<meta-data>` child element of the `<provider>` points to an XML file that specifies the directories you want to share. The `android:resource` attribute is the path and name of the file, without the `.xml` extension.The contents of this file are described in the next section.

Specify Sharable Directories

Once you have added the `FileProvider` to your app manifest, you need to specify the directories that contain the files you want to share. To specify the directories, start by creating the file `filepaths.xml` in the `res/xml/` subdirectory of your project. In this file, specify the directories by adding an XML element for each directory. The following snippet shows you an example of the contents of `res/xml/filepaths.xml`. The snippet also demonstrates how to share a subdirectory of the `files/`

directory in your internal storage area:

```
<paths>
    <files-path path="images/" name="myimages" />
</paths>
```

In this example, the `<files-path>` tag shares directories within the `files/` directory of your app's internal storage. The `path` attribute shares the `images/` subdirectory of `files/`. The `name` attribute tells the `FileProvider` to add the path segment `myimages` to content URIs for files in the `files/images/` subdirectory.

The `<paths>` element can have multiple children, each specifying a different directory to share. In addition to the `<files-path>` element, you can use the `<external-path>` element to share directories in external storage, and the `<cache-path>` element to share directories in your internal cache directory. To learn more about the child elements that specify shared directories, see the `FileProvider` reference documentation.

Note: The XML file is the only way you can specify the directories you want to share; you can't programmatically add a directory.

You now have a complete specification of a `FileProvider` that generates content URIs for files in the `files/` directory of your app's internal storage or for files in subdirectories of `files/`. When your app generates a content URI for a file, it contains the authority specified in the `<provider>` element (`com.example.myapp.fileprovider`), the path `myimages/`, and the name of the file.

For example, if you define a `FileProvider` according to the snippets in this lesson, and you request a content URI for the file `default_image.jpg`, `FileProvider` returns the following URI:

```
content://com.example.myapp.fileprovider/myimages/default_image.jpg
```

Next Get started Next class

Sharing a File

Previous Next Get started

This lesson teaches you to

1. Receive File Requests
2. Create a File Selection Activity
3. Respond to a File Selection
4. Grant Permissions for the File
5. Share the File with the Requesting App

You should also read

- Designing Content URIs
- Implementing Content Provider Permissions
- Permissions
- Intents and Intent Filters

Once you have set up your app to share files using content URIs, you can respond to other apps' requests for those files. One way to respond to these requests is to provide a file selection interface from the server app that other applications can invoke. This approach allows a client application to let users select a file from the server app and then receive the selected file's content URI.

This lesson shows you how to create a file selection `Activity` in your app that responds to requests for files.

Receive File Requests

To receive requests for files from client apps and respond with a content URI, your app should provide a file selection `Activity`. Client apps start

this Activity by calling startActivityForResult() with an Intent containing the action ACTION_PICK. When the client app calls startActivityForResult(), your app can return a result to the client app, in the form of a content URI for the file the user selected.

To learn how to implement a request for a file in a client app, see the lesson Requesting a Shared File.

Create a File Selection Activity

To set up the file selection Activity, start by specifying the Activity in your manifest, along with an intent filter that matches the action ACTION_PICK and the categories CATEGORY_DEFAULT and CATEGORY_OPENABLE. Also add MIME type filters for the files your app serves to other apps. The following snippet shows you how to specify the new Activity and intent filter:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android">
    ...
        <application>
        ...
            <activity
                android:name=".FileSelectActivity"
                android:label="@"File Selector" >
                <intent-filter>
                    <action
                        android:name="android.intent.action.PICK"/>
                    <category

android:name="android.intent.category.DEFAULT"/>
                    <category

android:name="android.intent.category.OPENABLE"/>
                    <data android:mimeType="text/plain"/>
                    <data android:mimeType="image/*"/>
                </intent-filter>
            </activity>
```

Define the file selection Activity in code

Next, define an Activity subclass that displays the files available from your app's `files/images/` directory in internal storage and allows the user to pick the desired file. The following snippet demonstrates how to define this Activity and respond to the user's selection:

```
public class MainActivity extends Activity {
    // The path to the root of this app's internal storage
    private File mPrivateRootDir;
    // The path to the "images" subdirectory
    private File mImagesDir;
    // Array of files in the images subdirectory
    File[] mImageFiles;
    // Array of filenames corresponding to mImageFiles
    String[] mImageFilenames;
    // Initialize the Activity
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        // Set up an Intent to send back to apps that request a file
        mResultIntent =
                new Intent("com.example.myapp.ACTION_RETURN_FILE");
        // Get the files/ subdirectory of internal storage
        mPrivateRootDir = getFilesDir();
        // Get the files/images subdirectory;
        mImagesDir = new File(mPrivateRootDir, "images");
        // Get the files in the images subdirectory
        mImageFiles = mImagesDir.listFiles();
        // Set the Activity's result to null to begin with
        setResult(Activity.RESULT_CANCELED, null);
        /*
         * Display the file names in the ListView mFileListView.
         * Back the ListView with the array mImageFilenames, which
         * you can create by iterating through mImageFiles and
         * calling File.getAbsolutePath() for each File
         */
         ...
    }
    ...
}
```

Respond to a File Selection

Once a user selects a shared file, your application must determine what file was selected and then generate a content URI for the file. Since the `Activity` displays the list of available files in a `ListView`, when the user clicks a file name the system calls the method `onItemClick()`, in which you can get the selected file.

In `onItemClick()`, get a `File` object for the file name of the selected file and pass it as an argument to `getUriForFile()`, along with the authority that you specified in the `<provider>` element for the `FileProvider`. The resulting content URI contains the authority, a path segment corresponding to the file's directory (as specified in the XML meta-data), and the name of the file including its extension. How `FileProvider` maps directories to path segments based on XML meta-data is described in the

section Specify Sharable Directories.

The following snippet shows you how to detect the selected file and get a content URI for it:

```
    protected void onCreate(Bundle savedInstanceState) {
        ...
        // Define a listener that responds to clicks on a file in
the ListView
        mFileListView.setOnItemClickListener(
                new AdapterView.OnItemClickListener() {
            @Override
            /*
             * When a filename in the ListView is clicked, get its
             * content URI and send it to the requesting app
             */
            public void onItemClick(AdapterView<?> adapterView,
                    View view,
                    int position,
                    long rowId) {
                /*
                 * Get a File for the selected file name.
                 * Assume that the file names are in the
                 * mImageFilename array.
                 */
                File requestFile = new
File(mImageFilename[position]);
                /*
                 * Most file-related method calls need to be in
                 * try-catch blocks.
                 */
                // Use the FileProvider to get a content URI
                try {
                    fileUri = FileProvider.getUriForFile(
                            MainActivity.this,
                            "com.example.myapp.fileprovider",
                            requestFile);
                } catch (IllegalArgumentException e) {
                    Log.e("File Selector",
                          "The selected file can't be shared: " +
                          clickedFilename);
                }
                ...
            }
        });
        ...
    }
```

Remember that you can only generate content URIs for files that reside in a directory you've specified in the meta-data file that contains the `<paths>` element, as described in the section Specify Sharable Directories. If you call `getUriForFile()` for a `File` in a path that you haven't specified, you receive an `IllegalArgumentException`.

Grant Permissions for the File

Now that you have a content URI for the file you want to share with another app, you need to allow the client app to access the file. To allow access, grant permissions to the client app by adding the content URI to an `Intent` and then setting permission flags on the `Intent`. The permissions you grant are temporary and expire automatically when the receiving app's task stack is finished.

The following code snippet shows you how to set read permission for the file:

```
    protected void onCreate(Bundle savedInstanceState) {
        ...
        // Define a listener that responds to clicks in the ListView
        mFileListView.setOnItemClickListener(
                new AdapterView.OnItemClickListener() {
            @Override
            public void onItemClick(AdapterView<?> adapterView,
                    View view,
                    int position,
                    long rowId) {
                ...
                if (fileUri != null) {
                    // Grant temporary read permission to the
content URI
                    mResultIntent.addFlags(
                        Intent.FLAG_GRANT_READ_URI_PERMISSION);
                }
                ...
            }
            ...
        });
    ...
    }
```

Caution: Calling `setFlags()` is the only way to securely grant access to your files using temporary access permissions. Avoid calling `Context.grantUriPermission()` method for a file's content URI, since this method grants access that you can only revoke by calling `Context.revokeUriPermission()`.

Share the File with the Requesting App

To share the file with the app that requested it, pass the `Intent` containing the content URI and permissions to `setResult()`. When the

Activity you have just defined is finished, the system sends the Intent containing the content URI to the client app. The following code snippet shows you how to do this:

```
    protected void onCreate(Bundle savedInstanceState) {
        ...
        // Define a listener that responds to clicks on a file in
the ListView
        mFileListView.setOnItemClickListener(
                new AdapterView.OnItemClickListener() {
            @Override
            public void onItemClick(AdapterView<?> adapterView,
                    View view,
                    int position,
                    long rowId) {
                ...
                if (fileUri != null) {
                    ...
                    // Put the Uri and MIME type in the result
Intent
                    mResultIntent.setDataAndType(
                            fileUri,
                            getContentResolver().getType(fileUri));
                    // Set the result
                    MainActivity.this.setResult(Activity.RESULT_OK,
                            mResultIntent);
                    } else {
                        mResultIntent.setDataAndType(null, "");
                        MainActivity.this.setResult(RESULT_CANCELED,
                                mResultIntent);
                    }
                }
        });
```

Provide users with an way to return immediately to the client app once they have chosen a file. One way to do this is to provide a checkmark or **Done** button. Associate a method with the button using the button's android:onClick attribute. In the method, call finish(). For example:

```
    public void onDoneClick(View v) {
        // Associate a method with the Done button
        finish();
    }
```

Next Get started Next class

Requesting a Shared File

Previous Next Get started

This lesson teaches you to

1. Send a Request for the File
2. Access the Requested File

You should also read

- Intents and Intent Filters
- Retrieving Data from the Provider

When an app wants to access a file shared by another app, the requesting app (the client) usually sends a request to the app sharing the files (the server). In most cases, the request starts an `Activity` in the server app that displays the files it can share. The user picks a file, after which the server app returns the file's content URI to the client app.

This lesson shows you how a client app requests a file from a server app, receives the file's content URI from the server app, and opens the file using the content URI.

Send a Request for the File

To request a file from the server app, the client app calls `startActivityForResult` with an `Intent` containing the action such as `ACTION_PICK` and a MIME type that the client app can handle.

For example, the following code snippet demonstrates how to send an `Intent` to a server app in order to start the `Activity` described in Sharing a File:

```
public class MainActivity extends Activity {
    private Intent mRequestFileIntent;
    private ParcelFileDescriptor mInputPFD;
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mRequestFileIntent = new Intent(Intent.ACTION_PICK);
        mRequestFileIntent.setType("image/jpg");
        ...
    }
    ...
    protected void requestFile() {
        /**
         * When the user requests a file, send an Intent to the
         * server app.
         * files.
         */
            startActivityForResult(mRequestFileIntent, 0);
        ...
    }
    ...
}
```

Access the Requested File

The server app sends the file's content URI back to the client app in an `Intent`. This `Intent` is passed to the client app in its override of `onActivityResult()`. Once the client app has the file's content URI, it can access the file by getting its `FileDescriptor`.

File security is preserved in this process because the content URI is the only piece of data that the client app receives. Since this URI doesn't contain a directory path, the client app can't discover and open any other files in the server app. Only the client app gets access to the file, and only for the permissions granted by the server app. The permissions are temporary, so once the client app's task stack is finished, the file is no longer accessible outside the server app.

The next snippet demonstrates how the client app handles the `Intent` sent from the server app, and how the client app gets the `FileDescriptor` using the content URI:

```
    /*
     * When the Activity of the app that hosts files sets a result
and calls
     * finish(), this method is invoked. The returned Intent
```

```
contains the
     * content URI of a selected file. The result code indicates if
the
     * selection worked or not.
     */
    @Override
    public void onActivityResult(int requestCode, int resultCode,
            Intent returnIntent) {
        // If the selection didn't work
        if (resultCode != RESULT_OK) {
            // Exit without doing anything else
            return;
        } else {
            // Get the file's content URI from the incoming Intent
            Uri returnUri = returnIntent.getData();
            /*
             * Try to open the file for "read" access using the
             * returned URI. If the file isn't found, write to the
             * error log and return.
             */
            try {
                /*
                 * Get the content resolver instance for this
context, and use it
                 * to get a ParcelFileDescriptor for the file.
                 */
                mInputPFD =
getContentResolver().openFileDescriptor(returnUri, "r");
            } catch (FileNotFoundException e) {
                e.printStackTrace();
                Log.e("MainActivity", "File not found.");
                return;
            }
            // Get a regular file descriptor for the file
            FileDescriptor fd = mInputPFD.getFileDescriptor();
            ...
        }
    }
```

The method `openFileDescriptor()` returns a `ParcelFileDescriptor` for the file. From this object, the client app gets a `FileDescriptor` object, which it can then use to read the file.

Next Get started Next class

Retrieving File Information

Previous Next Get started

This lesson teaches you to

1. Retrieve a File's MIME Type
2. Retrieve a File's Name and Size

You should also read

- Retrieving Data from the Provider

Before a client app tries to work with a file for which it has a content URI, the app can request information about the file from the server app, including the file's data type and file size. The data type helps the client app to determine if it can handle the file, and the file size helps the client app set up buffering and caching for the file.

This lesson demonstrates how to query the server app's `FileProvider` to retrieve a file's MIME type and size.

Retrieve a File's MIME Type

A file's data type indicates to the client app how it should handle the file's contents. To get the data type of a shared file given its content URI, the client app calls `ContentResolver.getType()`. This method returns the file's MIME type. By default, a `FileProvider` determines the file's MIME type from its filename extension.

The following code snippet demonstrates how a client app retrieves the MIME type of a file once the server app has returned the content URI to the client:

```
    ...
    /*
     * Get the file's content URI from the incoming Intent, then
     * get the file's MIME type
     */
    Uri returnUri = returnIntent.getData();
    String mimeType = getContentResolver().getType(returnUri);
    ...
```

Retrieve a File's Name and Size

The `FileProvider` class has a default implementation of the `query()` method that returns the name and size of the file associated with a content URI in a `Cursor`. The default implementation returns two columns:

`DISPLAY_NAME`
: The file's name, as a `String`. This value is the same as the value returned by `File.getName()`.

`SIZE`
: The size of the file in bytes, as a `long` This value is the same as the value returned by `File.length()`

The client app can get both the `DISPLAY_NAME` and `SIZE` for a file by setting all of the arguments of `query()` to `null` except for the content URI. For example, this code snippet retrieves a file's `DISPLAY_NAME` and `SIZE` and displays each one in separate `TextView`:

```
    ...
    /*
     * Get the file's content URI from the incoming Intent,
     * then query the server app to get the file's display name
     * and size.
     */
    Uri returnUri = returnIntent.getData();
    Cursor returnCursor =
            getContentResolver().query(returnUri, null, null, null,
null);
    /*
     * Get the column indexes of the data in the Cursor,
     * move to the first row in the Cursor, get the data,
     * and display it.
     */
    int nameIndex =
returnCursor.getColumnIndex(OpenableColumns.DISPLAY_NAME);
    int sizeIndex =
returnCursor.getColumnIndex(OpenableColumns.SIZE);
    returnCursor.moveToFirst();
```

```
    TextView nameView = (TextView) findViewById(R.id.filename_text);
    TextView sizeView = (TextView) findViewById(R.id.filesize_text);
    nameView.setText(returnCursor.getString(nameIndex));
    sizeView.setText(Long.toString(returnCursor.getLong(sizeIndex)));
    ...
```

Next Get started Next class

Sharing Files with NFC

Previous Next Get started

Dependencies and prerequisites

- Android 4.1 (API Level 16) or higher
- At least two NFC-enabled Android devices (NFC is not supported in the emulator)

You should also read

- Using the External Storage

Android allows you to transfer large files between devices using the Android Beam file transfer feature. This feature has a simple API and allows users to start the transfer process by simply touching devices. In response, Android Beam file transfer automatically copies files from one device to the other and notifies the user when it's finished.

While the Android Beam file transfer API handles large amounts of data, the Android Beam NDEF transfer API introduced in Android 4.0 (API level 14) handles small amounts of data such as URIs or other small messages. In addition, Android Beam is only one of the features available in the Android NFC framework, which allows you to read NDEF messages from NFC tags. To learn more about Android Beam, see the topic Beaming NDEF Messages to Other Devices. To learn more about the NFC framework, see the Near Field Communication API guide.

Lessons

Sending Files to Another Device

Learn how to set up your app to send files to another device.

Receiving Files from Another Device

Learn how to set up your app to receive files sent by another device.

Next Get started Next class

Sending Files to Another Device

Previous Next Get started

This lesson teaches you to

1. Declare Features in the Manifest
2. Test for Android Beam File Transfer Support
3. Create a Callback Method That Provides Files
4. Specify the Files to Send

You should also read

- Storage Options

This lesson shows you how to design your app to send large files to another device using Android Beam file transfer. To send files, you request permission to use NFC and external storage, test to ensure your device supports NFC, and provide URIs to Android Beam file transfer.

The Android Beam file transfer feature has the following requirements:

1. Android Beam file transfer for large files is only available in Android 4.1 (API level 16) and higher.
2. Files you want to transfer must reside in external storage. To learn more about using external storage, read Using the External Storage.
3. Each file you want to transfer must be world-readable. You can set this permission by calling the method `File.setReadable(true,false)`.
4. You must provide a file URI for the files you want to transfer. Android Beam file transfer is unable to handle content URIs generated by `FileProvider.getUriForFile`.

Declare Features in the Manifest

First, edit your app manifest to declare the permissions and features your app needs.

Request Permissions

To allow your app to use Android Beam file transfer to send files from external storage using NFC, you must request the following permissions in your app manifest:

NFC

Allows your app to send data over NFC. To specify this permission, add the following element as a child of the `<manifest>` element:

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

READ_EXTERNAL_STORAGE

Allows your app to read from external storage. To specify this permission, add the following element as a child of the `<manifest>` element:

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

Note: As of Android 4.2.2 (API level 17), this permission is not enforced. Future versions of the platform may require it for apps that want to read from external storage. To ensure forward compatibility, request the permission now, before it becomes required.

Specify the NFC feature

Specify that your app uses NFC, by adding a `<uses-feature>` element as a child of the `<manifest>` element. Set the `android:required` attribute to `true` to indicate that your app won't function unless NFC is present.

The following snippet shows you how to specify the `<uses-feature>` element:

```
<uses-feature
    android:name="android.hardware.nfc"
```

```
    android:required="true" />
```

Note that if your app only uses NFC as an option, but still functions if NFC isn't present, you should set `android:required` to `false`, and test for NFC in code.

Specify Android Beam file transfer

Since Android Beam file transfer is only available in Android 4.1 (API level 16) and later, if your app depends on Android Beam file transfer for a key part of its functionality you must specify the `<uses-sdk>` element with the `android:minSdkVersion="16"` attribute. Otherwise, you can set `android:minSdkVersion` to another value as necessary, and test for the platform version in code, as described in the following section.

Test for Android Beam File Transfer Support

To specify in your app manifest that NFC is optional, you use the following element:

```
<uses-feature android:name="android.hardware.nfc"
android:required="false" />
```

If you set the attribute `android:required="false"`, you must test for NFC support and Android Beam file transfer support in code.

To test for Android Beam file transfer support in code, start by testing that the device supports NFC by calling `PackageManager.hasSystemFeature()` with the argument `FEATURE_NFC`. Next, check that the Android version supports Android Beam file transfer by testing the value of `SDK_INT`. If Android Beam file transfer is supported, get an instance of the NFC controller, which allows you to communicate with the NFC hardware. For example:

```
public class MainActivity extends Activity {
    ...
    NfcAdapter mNfcAdapter;
    // Flag to indicate that Android Beam is available
    boolean mAndroidBeamAvailable  = false;
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
```

```
        ...
        // NFC isn't available on the device
        if
(!PackageManager.hasSystemFeature(PackageManager.FEATURE_NFC)) {
            /*
             * Disable NFC features here.
             * For example, disable menu items or buttons that
activate
             * NFC-related features
             */
            ...
        // Android Beam file transfer isn't supported
        } else if (Build.VERSION.SDK_INT <
                Build.VERSION_CODES.JELLY_BEAN_MR1) {
            // If Android Beam isn't available, don't continue.
            mAndroidBeamAvailable = false;
            /*
             * Disable Android Beam file transfer features here.
             */
            ...
        // Android Beam file transfer is available, continue
        } else {
        mNfcAdapter = NfcAdapter.getDefaultAdapter(this);
        ...
        }
    }
    ...
}
```

Create a Callback Method that Provides Files

Once you've verified that the device supports Android Beam file transfer, add a callback method that the system invokes when Android Beam file transfer detects that the user wants to send files to another NFC-enabled device. In this callback method, return an array of `Uri` objects. Android Beam file transfer copies the files represented by these URIs to the receiving device.

To add the callback method, implement the `NfcAdapter.CreateBeamUrisCallback` interface and its method `createBeamUris()`. The following snippet shows you how to do this:

```
public class MainActivity extends Activity {
    ...
    // List of URIs to provide to Android Beam
    private Uri[] mFileUris = new Uri[10];
    ...
    /**
     * Callback that Android Beam file transfer calls to get
     * files to share
```

```
     */
    private class FileUriCallback implements
            NfcAdapter.CreateBeamUrisCallback {
        public FileUriCallback() {
        }
        /**
         * Create content URIs as needed to share with another
device
         */
        @Override
        public Uri[] createBeamUris(NfcEvent event) {
            return mFileUris;
        }
    }
    ...
}
```

Once you've implemented the interface, provide the callback to Android Beam file transfer by calling `setBeamPushUrisCallback()`. The following snippet shows you how to do this:

```
public class MainActivity extends Activity {
    ...
    // Instance that returns available files from this app
    private FileUriCallback mFileUriCallback;
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        // Android Beam file transfer is available, continue
        ...
        mNfcAdapter = NfcAdapter.getDefaultAdapter(this);
        /*
         * Instantiate a new FileUriCallback to handle requests for
         * URIs
         */
        mFileUriCallback = new FileUriCallback();
        // Set the dynamic callback for URI requests.
        mNfcAdapter.setBeamPushUrisCallback(mFileUriCallback,this);
        ...
    }
    ...
}
```

Note: You can also provide the array of `Uri` objects directly to the NFC framework through your app's `NfcAdapter` instance. Choose this approach if you can define the URIs to transfer before the NFC touch event occurs. To learn more about this approach, see `NfcAdapter.setBeamPushUris()`.

Specify the Files to Send

To transfer one or more files to another NFC-enabled device, get a file

URI (a URI with a `file` scheme) for each file and then add the URI to an array of `Uri` objects. To transfer a file, you must also have permanent read access for the file. For example, the following snippet shows you how to get a file URI from a file name and then add the URI to the array:

```
/*
 * Create a list of URIs, get a File,
 * and set its permissions
 */
private Uri[] mFileUris = new Uri[10];
String transferFile = "transferimage.jpg";
File extDir = getExternalFilesDir(null);
File requestFile = new File(extDir, transferFile);
requestFile.setReadable(true, false);
// Get a URI for the File and add it to the list of URIs
fileUri = Uri.fromFile(requestFile);
if (fileUri != null) {
    mFileUris[0] = fileUri;
} else {
    Log.e("My Activity", "No File URI available for file.");
}
```

Next Get started Next class

Receiving Files from Another Device

Previous Next Get started

This lesson teaches you to

1. Respond to a Request to Display Data
2. Request File Permissions
3. Get the Directory for Copied Files

You should also read

- Content URIs
- Intents and Intent Filters
- Notifications
- Using the External Storage

Android Beam file transfer copies files to a special directory on the receiving device. It also scans the copied files using the Android Media Scanner and adds entries for media files to the `MediaStore` provider. This lesson shows you how to respond when the file copy is complete, and how to locate the copied files on the receiving device.

Respond to a Request to Display Data

When Android Beam file transfer finishes copying files to the receiving device, it posts a notification containing an `Intent` with the action `ACTION_VIEW`, the MIME type of the first file that was transferred, and a URI that points to the first file. When the user clicks the notification, this intent is sent out to the system. To have your app respond to this intent, add an `<intent-filter>` element for the `<activity>` element of the

Activity that should respond. In the <intent-filter> element, add the following child elements:

<action android:name="android.intent.action.VIEW" />
: Matches the ACTION_VIEW intent sent from the notification.

<category android:name="android.intent.category.CATEGORY_DEFAULT" />
: Matches an Intent that doesn't have an explicit category.

<data android:mimeType="*mime-type*" />
: Matches a MIME type. Specify only those MIME types that your app can handle.

For example, the following snippet shows you how to add an intent filter that triggers the activity `com.example.android.nfctransfer.ViewActivity`:

```
    <activity
        android:name="com.example.android.nfctransfer.ViewActivity"
        android:label="Android Beam Viewer" >
        ...
        <intent-filter>
            <action android:name="android.intent.action.VIEW"/>
            <category
android:name="android.intent.category.DEFAULT"/>
            ...
        </intent-filter>
    </activity>
```

Note: Android Beam file transfer is not the only source of an ACTION_VIEW intent. Other apps on the receiving device can also send an Intent with this action. Handling this situation is discussed in the section Get the directory from a content URI.

Request File Permissions

To read files that Android Beam file transfer copies to the device, request the permission READ_EXTERNAL_STORAGE. For example:

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

If you want to copy transferred files to your app's own storage area, request the permission WRITE_EXTERNAL_STORAGE instead. WRITE_EXTERNAL_STORAGE includes READ_EXTERNAL_STORAGE.

Note: As of Android 4.2.2 (API level 17), the permission READ_EXTERNAL_STORAGE is only enforced if the user chooses to do so. Future versions of the platform may require this permission in all cases. To ensure forward compatibility, request the permission now, before it becomes required.

Since your app has control over its internal storage area, you don't need to request write permission to copy a transferred file to your internal storage area.

Get the Directory for Copied Files

Android Beam file transfer copies all the files in a single transfer to one directory on the receiving device. The URI in the content Intent sent by the Android Beam file transfer notification points to the first transferred file. However, your app may also receive an ACTION_VIEW intent from a source other than Android Beam file transfer. To determine how you should handle the incoming Intent, you need to examine its scheme and authority.

To get the scheme for the URI, call Uri.getScheme(). The following code snippet shows you how to determine the scheme and handle the URI accordingly:

```
public class MainActivity extends Activity {
    ...
    // A File object containing the path to the transferred files
    private File mParentPath;
    // Incoming Intent
    private Intent mIntent;
    ...
    /*
     * Called from onNewIntent() for a SINGLE_TOP Activity
     * or onCreate() for a new Activity. For onNewIntent(),
     * remember to call setIntent() to store the most
     * current Intent
     *
     */
    private void handleViewIntent() {
        ...
        // Get the Intent action
        mIntent = getIntent();
        String action = mIntent.getAction();
        /*
         * For ACTION_VIEW, the Activity is being asked to display
data.
```

```
         * Get the URI.
         */
        if (TextUtils.equals(action, Intent.ACTION_VIEW)) {
            // Get the URI from the Intent
            Uri beamUri = mIntent.getData();
            /*
             * Test for the type of URI, by getting its scheme value
             */
            if (TextUtils.equals(beamUri.getScheme(), "file")) {
                mParentPath = handleFileUri(beamUri);
            } else if (TextUtils.equals(
                    beamUri.getScheme(), "content")) {
                mParentPath = handleContentUri(beamUri);
            }
        }
        ...
    }
    ...
}
```

Get the directory from a file URI

If the incoming `Intent` contains a file URI, the URI contains the absolute file name of a file, including the full directory path and file name. For Android Beam file transfer, the directory path points to the location of the other transferred files, if any. To get the directory path, get the path part of the URI, which contains all of the URI except the `file:` prefix. Create a `File` from the path part, then get the parent path of the `File`:

```
    ...
    public String handleFileUri(Uri beamUri) {
        // Get the path part of the URI
        String fileName = beamUri.getPath();
        // Create a File object for this filename
        File copiedFile = new File(fileName);
        // Get a string containing the file's parent directory
        return copiedFile.getParent();
    }
    ...
```

Get the directory from a content URI

If the incoming `Intent` contains a content URI, the URI may point to a directory and file name stored in the `MediaStore` content provider. You can detect a content URI for `MediaStore` by testing the URI's authority value. A content URI for `MediaStore` may come from Android Beam file transfer or from another app, but in both cases you can retrieve a directory and file name for the content URI.

You can also receive an incoming `ACTION_VIEW` intent containing a content URI for a content provider other than `MediaStore`. In this case, the content URI doesn't contain the `MediaStore` authority value, and the content URI usually doesn't point to a directory.

Note: For Android Beam file transfer, you receive a content URI in the `ACTION_VIEW` intent if the first incoming file has a MIME type of "audio/*", "image/*", or "video/*", indicating that the file is media- related. Android Beam file transfer indexes the media files it transfers by running Media Scanner on the directory where it stores transferred files. Media Scanner writes its results to the `MediaStore` content provider, then it passes a content URI for the first file back to Android Beam file transfer. This content URI is the one you receive in the notification `Intent`. To get the directory of the first file, you retrieve it from `MediaStore` using the content URI.

Determine the content provider

To determine if you can retrieve a file directory from the content URI, determine the the content provider associated with the URI by calling `Uri.getAuthority()` to get the URI's authority. The result has two possible values:

`MediaStore.AUTHORITY`
: The URI is for a file or files tracked by `MediaStore`. Retrieve the full file name from `MediaStore`, and get directory from the file name.

Any other authority value
: A content URI from another content provider. Display the data associated with the content URI, but don't get the file directory.

To get the directory for a `MediaStore` content URI, run a query that specifies the incoming content URI for the `Uri` argument and the column `MediaColumns.DATA` for the projection. The returned `Cursor` contains the full path and name for the file represented by the URI. This path also contains all the other files that Android Beam file transfer just copied to the device.

The following snippet shows you how to test the authority of the content

URI and retrieve the the path and file name for the transferred file:

```
    ...
    public String handleContentUri(Uri beamUri) {
        // Position of the filename in the query Cursor
        int filenameIndex;
        // File object for the filename
        File copiedFile;
        // The filename stored in MediaStore
        String fileName;
        // Test the authority of the URI
        if (!TextUtils.equals(beamUri.getAuthority(),
MediaStore.AUTHORITY)) {
            /*
             * Handle content URIs for other content providers
             */
        // For a MediaStore content URI
        } else {
            // Get the column that contains the file name
            String[] projection = { MediaStore.MediaColumns.DATA };
            Cursor pathCursor =
                    getContentResolver().query(beamUri, projection,
                    null, null, null);
            // Check for a valid cursor
            if (pathCursor != null &&
                    pathCursor.moveToFirst()) {
                // Get the column index in the Cursor
                filenameIndex = pathCursor.getColumnIndex(
                        MediaStore.MediaColumns.DATA);
                // Get the full file name including path
                fileName = pathCursor.getString(filenameIndex);
                // Create a File object for the filename
                copiedFile = new File(fileName);
                // Return the parent directory of the file
                return new File(copiedFile.getParent());
            } else {
                // The query didn't work; return null
                return null;
            }
        }
    }
    ...
```

To learn more about retrieving data from a content provider, see the section Retrieving Data from the Provider.

Next Get started Next class

Building Apps with Multimedia

These classes teach you how to create rich multimedia apps that behave the way users expect.

Managing Audio Playback

Previous Next Get started

Dependencies and prerequisites

- Android 2.0 (API level 5) or higher
- Experience with Media Playback

You should also read

- Services

If your app plays audio, it’s important that your users can control the audio in a predictable manner. To ensure a great user experience, it’s also important that your app manages the audio focus to ensure multiple apps aren’t playing audio at the same time.

After this class, you will be able to build apps that respond to hardware audio key presses, which request audio focus when playing audio, and which respond appropriately to changes in audio focus caused by the system or other applications.

Lessons

Controlling Your App’s Volume and Playback

Learn how to ensure your users can control the volume of your app using the hardware or software volume controls and where available the play, stop, pause, skip, and previous media playback keys.

Managing Audio Focus

With multiple apps potentially playing audio it's important to think about how they should interact. To avoid every music app playing at the same time, Android uses audio focus to moderate audio

playback. Learn how to request the audio focus, listen for a loss of audio focus, and how to respond when that happens.

Dealing with Audio Output Hardware

Audio can be played from a number of sources. Learn how to find out where the audio is being played and how to handle a headset being disconnected during playback.

Next Get started Next class

Controlling Your App's Volume and Playback

Previous Next Get started

This lesson teaches you to

1. Identify Which Audio Stream to Use
2. Use Hardware Volume Keys to Control Your App's Audio Volume
3. Use Hardware Playback Control Keys to Control Your App's Audio Playback

You should also read

- Media Playback

A good user experience is a predictable one. If your app plays media it's important that your users can control the volume of your app using the hardware or software volume controls of their device, bluetooth headset, or headphones.

Similarly, where appropriate and available, the play, stop, pause, skip, and previous media playback keys should perform their respective actions on the audio stream used by your app.

Identify Which Audio Stream to Use

The first step to creating a predictable audio experience is understanding which audio stream your app will use.

Android maintains a separate audio stream for playing music, alarms, notifications, the incoming call ringer, system sounds, in-call volume, and

DTMF tones. This is done primarily to allow users to control the volume of each stream independently.

Most of these streams are restricted to system events, so unless your app is a replacement alarm clock, you'll almost certainly be playing your audio using the `STREAM_MUSIC` stream.

Use Hardware Volume Keys to Control Your App's Audio Volume

By default, pressing the volume controls modify the volume of the active audio stream. If your app isn't currently playing anything, hitting the volume keys adjusts the ringer volume.

If you've got a game or music app, then chances are good that when the user hits the volume keys they want to control the volume of the game or music, even if they're currently between songs or there's no music in the current game location.

You may be tempted to try and listen for volume key presses and modify the volume of your audio stream that way. Resist the urge. Android provides the handy `setVolumeControlStream()` method to direct volume key presses to the audio stream you specify.

Having identified the audio stream your application will be using, you should set it as the volume stream target. You should make this call early in your app's lifecycle—because you only need to call it once during the activity lifecycle, you should typically call it within the `onCreate()` method (of the `Activity` or `Fragment` that controls your media). This ensures that whenever your app is visible, the volume controls function as the user expects.

```
setVolumeControlStream(AudioManager.STREAM_MUSIC);
```

From this point onwards, pressing the volume keys on the device affect the audio stream you specify (in this case "music") whenever the target activity or fragment is visible.

Use Hardware Playback Control Keys to Control Your App's Audio Playback

Media playback buttons such as play, pause, stop, skip, and previous are available on some handsets and many connected or wireless headsets. Whenever a user presses one of these hardware keys, the system broadcasts an intent with the `ACTION_MEDIA_BUTTON` action.

To respond to media button clicks, you need to register a `BroadcastReceiver` in your manifest that listens for this action broadcast as shown below.

```
<receiver android:name=".RemoteControlReceiver">
    <intent-filter>
        <action android:name="android.intent.action.MEDIA_BUTTON" />
    </intent-filter>
</receiver>
```

The receiver implementation itself needs to extract which key was pressed to cause the broadcast. The `Intent` includes this under the `EXTRA_KEY_EVENT` key, while the `KeyEvent` class includes a list `KEYCODE_MEDIA_*` static constants that represents each of the possible media buttons, such as `KEYCODE_MEDIA_PLAY_PAUSE` and `KEYCODE_MEDIA_NEXT`.

The following snippet shows how to extract the media button pressed and affects the media playback accordingly.

```
public class RemoteControlReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        if (Intent.ACTION_MEDIA_BUTTON.equals(intent.getAction())) {
            KeyEvent event =
(KeyEvent)intent.getParcelableExtra(Intent.EXTRA_KEY_EVENT);
            if (KeyEvent.KEYCODE_MEDIA_PLAY == event.getKeyCode()) {
                // Handle key press.
            }
        }
    }
}
```

Because multiple applications might want to listen for media button presses, you must also programmatically control when your app should receive media button press events.

The following code can be used within your app to register and de-register your media button event receiver using the `AudioManager`. When registered, your broadcast receiver is the exclusive receiver of all media button broadcasts.

```
AudioManager am = mContext.getSystemService(Context.AUDIO_SERVICE);
...

// Start listening for button presses
am.registerMediaButtonEventReceiver(RemoteControlReceiver);
...

// Stop listening for button presses
am.unregisterMediaButtonEventReceiver(RemoteControlReceiver);
```

Typically, apps should unregister most of their receivers whenever they become inactive or invisible (such as during the `onStop()` callback). However, it's not that simple for media playback apps—in fact, responding to media playback buttons is most important when your application isn't visible and therefore can't be controlled by the on-screen UI.

A better approach is to register and unregister the media button event receiver when your application gains and loses the audio focus. This is covered in detail in the next lesson.

Next Get started Next class

Managing Audio Focus

Previous Next Get started

This lesson teaches you to

1. Request the Audio Focus
2. Handle the Loss of Audio Focus
3. Duck!

You should also read

- Media Playback

With multiple apps potentially playing audio it's important to think about how they should interact. To avoid every music app playing at the same time, Android uses audio focus to moderate audio playback—only apps that hold the audio focus should play audio.

Before your app starts playing audio it should request—and receive—the audio focus. Likewise, it should know how to listen for a loss of audio focus and respond appropriately when that happens.

Request the Audio Focus

Before your app starts playing any audio, it should hold the audio focus for the stream it will be using. This is done with a call to `requestAudioFocus()` which returns `AUDIOFOCUS_REQUEST_GRANTED` if your request is successful.

You must specify which stream you're using and whether you expect to require transient or permanent audio focus. Request transient focus when you expect to play audio for only a short time (for example when playing

navigation instructions). Request permanent audio focus when you plan to play audio for the foreseeable future (for example, when playing music).

The following snippet requests permanent audio focus on the music audio stream. You should request the audio focus immediately before you begin playback, such as when the user presses play or the background music for the next game level begins.

```
AudioManager am = mContext.getSystemService(Context.AUDIO_SERVICE);
...

// Request audio focus for playback
int result = am.requestAudioFocus(afChangeListener,
                                  // Use the music stream.
                                  AudioManager.STREAM_MUSIC,
                                  // Request permanent focus.
                                  AudioManager.AUDIOFOCUS_GAIN);

if (result == AudioManager.AUDIOFOCUS_REQUEST_GRANTED) {
    am.registerMediaButtonEventReceiver(RemoteControlReceiver);
    // Start playback.
}
```

Once you've finished playback be sure to call `abandonAudioFocus()`. This notifies the system that you no longer require focus and unregisters the associated `AudioManager.OnAudioFocusChangeListener`. In the case of abandoning transient focus, this allows any interupted app to continue playback.

```
// Abandon audio focus when playback complete
am.abandonAudioFocus(afChangeListener);
```

When requesting transient audio focus you have an additional option: whether or not you want to enable "ducking." Normally, when a well-behaved audio app loses audio focus it immediately silences its playback. By requesting a transient audio focus that allows ducking you tell other audio apps that it’s acceptable for them to keep playing, provided they lower their volume until the focus returns to them.

```
// Request audio focus for playback
int result = am.requestAudioFocus(afChangeListener,
                                  // Use the music stream.
                                  AudioManager.STREAM_MUSIC,
                                  // Request permanent focus.

AudioManager.AUDIOFOCUS_GAIN_TRANSIENT_MAY_DUCK);
```

```
if (result == AudioManager.AUDIOFOCUS_REQUEST_GRANTED) {
    // Start playback.
}
```

Ducking is particularly suitable for apps that use the audio stream intermittently, such as for audible driving directions.

Whenever another app requests audio focus as described above, its choice between permanent and transient (with or without support for ducking) audio focus is received by the listener you registered when requesting focus.

Handle the Loss of Audio Focus

If your app can request audio focus, it follows that it will in turn lose that focus when another app requests it. How your app responds to a loss of audio focus depends on the manner of that loss.

The `onAudioFocusChange()` callback method of the audio focus change listener you registered when requesting audio focus receives a parameter that describes the focus change event. Specifically, the possible focus loss events mirror the focus request types from the previous section—permanent loss, transient loss, and transient with ducking permitted.

Generally speaking, a transient (temporary) loss of audio focus should result in your app silencing it's audio stream, but otherwise maintaining the same state. You should continue to monitor changes in audio focus and be prepared to resume playback where it was paused once you've regained the focus.

If the audio focus loss is permanent, it's assumed that another application is now being used to listen to audio and your app should effectively end itself. In practical terms, that means stopping playback, removing media button listeners—allowing the new audio player to exclusively handle those events—and abandoning your audio focus. At that point, you would expect a user action (pressing play in your app) to be required before you resume playing audio.

In the following code snippet, we pause the playback or our media player object if the audio loss is transient and resume it when we have regained the focus. If the loss is permanent, it unregisters our media button event receiver and stops monitoring audio focus changes.

```
OnAudioFocusChangeListener afChangeListener = new
OnAudioFocusChangeListener() {
    public void onAudioFocusChange(int focusChange) {
        if (focusChange == AUDIOFOCUS_LOSS_TRANSIENT
            // Pause playback
        } else if (focusChange == AudioManager.AUDIOFOCUS_GAIN) {
            // Resume playback
        } else if (focusChange == AudioManager.AUDIOFOCUS_LOSS) {

am.unregisterMediaButtonEventReceiver(RemoteControlReceiver);
            am.abandonAudioFocus(afChangeListener);
            // Stop playback
        }
    }
};
```

In the case of a transient loss of audio focus where ducking is permitted, rather than pausing playback, you can "duck" instead.

Duck!

Ducking is the process of lowering your audio stream output volume to make transient audio from another app easier to hear without totally disrupting the audio from your own application.

In the following code snippet lowers the volume on our media player object when we temporarily lose focus, then returns it to its previous level when we regain focus.

```
OnAudioFocusChangeListener afChangeListener = new
OnAudioFocusChangeListener() {
    public void onAudioFocusChange(int focusChange) {
        if (focusChange == AUDIOFOCUS_LOSS_TRANSIENT_CAN_DUCK) {
            // Lower the volume
        } else if (focusChange == AudioManager.AUDIOFOCUS_GAIN) {
            // Raise it back to normal
        }
    }
};
```

A loss of audio focus is the most important broadcast to react to, but not the only one. The system broadcasts a number of intents to alert you to

changes in user's audio experience. The next lesson demonstrates how to monitor them to improve the user's overall experience.

Next Get started Next class

Dealing with Audio Output Hardware

Previous Next Get started

This lesson teaches you to

1. Check What Hardware is Being Used
2. Handle Changes in the Audio Output Hardware

You should also read

- Media Playback

Users have a number of alternatives when it comes to enjoying the audio from their Android devices. Most devices have a built-in speaker, headphone jacks for wired headsets, and many also feature Bluetooth connectivity and support for A2DP audio.

Check What Hardware is Being Used

How your app behaves might be affected by which hardware its output is being routed to.

You can query the `AudioManager` to determine if the audio is currently being routed to the device speaker, wired headset, or attached Bluetooth device as shown in the following snippet:

```
if (isBluetoothA2dpOn()) {
    // Adjust output for Bluetooth.
} else if (isSpeakerphoneOn()) {
    // Adjust output for Speakerphone.
} else if (isWiredHeadsetOn()) {
    // Adjust output for headsets
} else {
```

```
    // If audio plays and noone can hear it, is it still playing?
}
```

Handle Changes in the Audio Output Hardware

When a headset is unplugged, or a Bluetooth device disconnected, the audio stream automatically reroutes to the built in speaker. If you listen to your music at as high a volume as I do, that can be a noisy surprise.

Luckily the system broadcasts an `ACTION_AUDIO_BECOMING_NOISY` intent when this happens. It's good practice to register a `BroadcastReceiver` that listens for this intent whenever you're playing audio. In the case of music players, users typically expect the playback to be paused—while for games you may choose to significantly lower the volume.

```
private class NoisyAudioStreamReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        if
(AudioManager.ACTION_AUDIO_BECOMING_NOISY.equals(intent.getAction()))
{
            // Pause the playback
        }
    }
}

private IntentFilter intentFilter = new
IntentFilter(AudioManager.ACTION_AUDIO_BECOMING_NOISY);

private void startPlayback() {
    registerReceiver(myNoisyAudioStreamReceiver(), intentFilter);
}

private void stopPlayback() {
    unregisterReceiver(myNoisyAudioStreamReceiver);
}
```

Next Get started Next class

Capturing Photos

Previous Next Get started

Dependencies and prerequisites

- Android 2.2 (API level 8) or higher
- A device with a camera

You should also read

- Camera
- Activities

Try it out

Download the sample

PhotoIntentActivity.zip

The world was a dismal and featureless place before rich media became prevalent. Remember Gopher? We don't, either. For your app to become part of your users' lives, give them a way to put their lives into it. Using the on-board cameras, your application can enable users to augment what they see around them, make unique avatars, look for zombies around the corner, or simply share their experiences.

This class gets you clicking fast with some super-easy ways of leveraging existing camera applications. In later lessons, you dive deeper and learn how to control the camera hardware directly.

Lessons

Taking Photos Simply

Leverage other applications and capture photos with just a few lines of code.

Recording Videos Simply

Leverage other applications and record videos with just a few lines of code.

Controlling the Camera

Control the camera hardware directly and implement your own camera application.

Next Get started Next class

Taking Photos Simply

Previous Next Get started

This lesson teaches you to

1. Request Camera Permission
2. Take a Photo with the Camera App
3. Get the Thumbnail
4. Save the Full-size Photo
5. Add the Photo to a Gallery
6. Decode a Scaled Image

You should also read

- Camera
- Intents and Intent Filters

Try it out

Download the sample

PhotoIntentActivity.zip

This lesson explains how to capture photos using an existing camera application.

Suppose you are implementing a crowd-sourced weather service that makes a global weather map by blending together pictures of the sky taken by devices running your client app. Integrating photos is only a small part of your application. You want to take photos with minimal fuss, not reinvent the camera. Happily, most Android-powered devices already have at least one camera application installed. In this lesson, you

learn how to make it take a picture for you.

Request Camera Permission

If an essential function of your application is taking pictures, then restrict its visibility on Google Play to devices that have a camera. To advertise that your application depends on having a camera, put a `<uses-feature>` tag in your manifest file:

```
<manifest ... >
    <uses-feature android:name="android.hardware.camera"
                  android:required="true" />
    ...
</manifest>
```

If your application uses, but does not require a camera in order to function, instead set `android:required` to `false`. In doing so, Google Play will allow devices without a camera to download your application. It's then your responsibility to check for the availability of the camera at runtime by calling `hasSystemFeature(PackageManager.FEATURE_CAMERA)`. If a camera is not available, you should then disable your camera features.

Take a Photo with the Camera App

The Android way of delegating actions to other applications is to invoke an `Intent` that describes what you want done. This process involves three pieces: The `Intent` itself, a call to start the external `Activity`, and some code to handle the image data when focus returns to your activity.

Here's a function that invokes an intent to capture a photo.

```
static final int REQUEST_IMAGE_CAPTURE = 1;

private void dispatchTakePictureIntent() {
    Intent takePictureIntent = new
Intent(MediaStore.ACTION_IMAGE_CAPTURE);
    if (takePictureIntent.resolveActivity(getPackageManager()) !=
null) {
        startActivityForResult(takePictureIntent,
REQUEST_IMAGE_CAPTURE);
    }
}
```

Notice that the `startActivityForResult()` method is protected by a condition that calls `resolveActivity()`, which returns the first activity component that can handle the intent. Performing this check is important because if you call `startActivityForResult()` using an intent that no app can handle, your app will crash. So as long as the result is not null, it's safe to use the intent.

Get the Thumbnail

If the simple feat of taking a photo is not the culmination of your app's ambition, then you probably want to get the image back from the camera application and do something with it.

The Android Camera application encodes the photo in the return `Intent` delivered to `onActivityResult()` as a small `Bitmap` in the extras, under the key `"data"`. The following code retrieves this image and displays it in an `ImageView`.

```
@Override
protected void onActivityResult(int requestCode, int resultCode,
Intent data) {
    if (requestCode == REQUEST_IMAGE_CAPTURE && resultCode ==
RESULT_OK) {
        Bundle extras = data.getExtras();
        Bitmap imageBitmap = (Bitmap) extras.get("data");
        mImageView.setImageBitmap(imageBitmap);
    }
}
```

Note: This thumbnail image from `"data"` might be good for an icon, but not a lot more. Dealing with a full-sized image takes a bit more work.

Save the Full-size Photo

The Android Camera application saves a full-size photo if you give it a file to save into. You must provide a fully qualified file name where the camera app should save the photo.

Generally, any photos that the user captures with the device camera should be saved on the device in the public external storage so they are accessible by all apps. The proper directory for shared photos is provided

by getExternalStoragePublicDirectory(), with the DIRECTORY_PICTURES argument. Because the directory provided by this method is shared among all apps, reading and writing to it requires the READ_EXTERNAL_STORAGE and WRITE_EXTERNAL_STORAGE permissions, respectively. The write permission implicitly allows reading, so if you need to write to the external storage then you need to request only one permission:

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

However, if you'd like the photos to remain private to your app only, you can instead use the directory provided by getExternalFilesDir(). On Android 4.3 and lower, writing to this directory also requires the WRITE_EXTERNAL_STORAGE permission. Beginning with Android 4.4, the permission is no longer required because the directory is not accessible by other apps, so you can declare the permission should be requested only on the lower versions of Android by adding the maxSdkVersion attribute:

```
<manifest ...>
    <uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE"
                     android:maxSdkVersion="18" />
    ...
</manifest>
```

Note: Files you save in the directories provided by getExternalFilesDir() are deleted when the user uninstalls your app.

Once you decide the directory for the file, you need to create a collision-resistant file name. You may wish also to save the path in a member variable for later use. Here's an example solution in a method that returns a unique file name for a new photo using a date-time stamp:

```
String mCurrentPhotoPath;

private File createImageFile() throws IOException {
    // Create an image file name
    String timeStamp = new
SimpleDateFormat("yyyyMMdd_HHmmss").format(new Date());
    String imageFileName = "JPEG_" + timeStamp + "_";
    File storageDir = Environment.getExternalStoragePublicDirectory(
```

```
            Environment.DIRECTORY_PICTURES);
    File image = File.createTempFile(
        imageFileName,  /* prefix */
        ".jpg",         /* suffix */
        storageDir      /* directory */
    );

    // Save a file: path for use with ACTION_VIEW intents
    mCurrentPhotoPath = "file:" + image.getAbsolutePath();
    return image;
}
```

With this method available to create a file for the photo, you can now create and invoke the `Intent` like this:

```
static final int REQUEST_TAKE_PHOTO = 1;

private void dispatchTakePictureIntent() {
    Intent takePictureIntent = new
Intent(MediaStore.ACTION_IMAGE_CAPTURE);
    // Ensure that there's a camera activity to handle the intent
    if (takePictureIntent.resolveActivity(getPackageManager()) !=
null) {
        // Create the File where the photo should go
        File photoFile = null;
        try {
            photoFile = createImageFile();
        } catch (IOException ex) {
            // Error occurred while creating the File
            ...
        }
        // Continue only if the File was successfully created
        if (photoFile != null) {
            takePictureIntent.putExtra(MediaStore.EXTRA_OUTPUT,
                    Uri.fromFile(photoFile));
            startActivityForResult(takePictureIntent,
REQUEST_TAKE_PHOTO);
        }
    }
}
```

Add the Photo to a Gallery

When you create a photo through an intent, you should know where your image is located, because you said where to save it in the first place. For everyone else, perhaps the easiest way to make your photo accessible is to make it accessible from the system's Media Provider.

Note: If you saved your photo to the directory provided by `getExternalFilesDir()`, the media scanner cannot access the files because they are private to your app.

The following example method demonstrates how to invoke the system's media scanner to add your photo to the Media Provider's database, making it available in the Android Gallery application and to other apps.

```
private void galleryAddPic() {
    Intent mediaScanIntent = new
Intent(Intent.ACTION_MEDIA_SCANNER_SCAN_FILE);
    File f = new File(mCurrentPhotoPath);
    Uri contentUri = Uri.fromFile(f);
    mediaScanIntent.setData(contentUri);
    this.sendBroadcast(mediaScanIntent);
}
```

Decode a Scaled Image

Managing multiple full-sized images can be tricky with limited memory. If you find your application running out of memory after displaying just a few images, you can dramatically reduce the amount of dynamic heap used by expanding the JPEG into a memory array that's already scaled to match the size of the destination view. The following example method demonstrates this technique.

```
private void setPic() {
    // Get the dimensions of the View
    int targetW = mImageView.getWidth();
    int targetH = mImageView.getHeight();

    // Get the dimensions of the bitmap
    BitmapFactory.Options bmOptions = new BitmapFactory.Options();
    bmOptions.inJustDecodeBounds = true;
    BitmapFactory.decodeFile(mCurrentPhotoPath, bmOptions);
    int photoW = bmOptions.outWidth;
    int photoH = bmOptions.outHeight;

    // Determine how much to scale down the image
    int scaleFactor = Math.min(photoW/targetW, photoH/targetH);

    // Decode the image file into a Bitmap sized to fill the View
    bmOptions.inJustDecodeBounds = false;
    bmOptions.inSampleSize = scaleFactor;
    bmOptions.inPurgeable = true;

    Bitmap bitmap = BitmapFactory.decodeFile(mCurrentPhotoPath,
bmOptions);
    mImageView.setImageBitmap(bitmap);
}
```

Next Get started Next class

Recording Videos Simply

Previous Next Get started

This lesson teaches you to

1. Request Camera Permission
2. Record a Video with a Camera App
3. View the Video

You should also read

- Camera
- Intents and Intent Filters

Try it out

Download the sample

PhotoIntentActivity.zip

This lesson explains how to capture video using existing camera applications.

Your application has a job to do, and integrating videos is only a small part of it. You want to take videos with minimal fuss, and not reinvent the camcorder. Happily, most Android-powered devices already have a camera application that records video. In this lesson, you make it do this for you.

Request Camera Permission

To advertise that your application depends on having a camera, put a

`<uses-feature>` tag in the manifest file:

```
<manifest ... >
    <uses-feature android:name="android.hardware.camera"
                  android:required="true" />
    ...
</manifest>
```

If your application uses, but does not require a camera in order to function, set `android:required` to `false`. In doing so, Google Play will allow devices without a camera to download your application. It's then your responsibility to check for the availability of the camera at runtime by calling `hasSystemFeature(PackageManager.FEATURE_CAMERA)`. If a camera is not available, you should then disable your camera features.

Record a Video with a Camera App

The Android way of delegating actions to other applications is to invoke an `Intent` that describes what you want done. This process involves three pieces: The `Intent` itself, a call to start the external `Activity`, and some code to handle the video when focus returns to your activity.

Here's a function that invokes an intent to capture video.

```
static final int REQUEST_VIDEO_CAPTURE = 1;

private void dispatchTakeVideoIntent() {
    Intent takeVideoIntent = new
Intent(MediaStore.ACTION_VIDEO_CAPTURE);
    if (takeVideoIntent.resolveActivity(getPackageManager()) != null)
{
        startActivityForResult(takeVideoIntent,
REQUEST_VIDEO_CAPTURE);
    }
}
```

Notice that the `startActivityForResult()` method is protected by a condition that calls `resolveActivity()`, which returns the first activity component that can handle the intent. Performing this check is important because if you call `startActivityForResult()` using an intent that no app can handle, your app will crash. So as long as the result is not null, it's safe to use the intent.

View the Video

The Android Camera application returns the video in the `Intent` delivered to `onActivityResult()` as a `Uri` pointing to the video location in storage. The following code retrieves this video and displays it in a `VideoView`.

```
@Override
protected void onActivityResult(int requestCode, int resultCode,
Intent data) {
    if (requestCode == REQUEST_VIDEO_CAPTURE && resultCode ==
RESULT_OK) {
        Uri videoUri = intent.getData();
        mVideoView.setVideoURI(videoUri);
    }
}
```

Next Get started Next class

Controlling the Camera

Previous Next Get started

This lesson teaches you to

1. Open the Camera Object
2. Create the Camera Preview
3. Modify Camera Settings
4. Set the Preview Orientation
5. Take a Picture
6. Restart the Preview
7. Stop the Preview and Release the Camera

You should also read

- Building a Camera App

In this lesson, we discuss how to control the camera hardware directly using the framework APIs.

Directly controlling a device camera requires a lot more code than requesting pictures or videos from existing camera applications. However, if you want to build a specialized camera application or something fully integrated in your app UI, this lesson shows you how.

Open the Camera Object

Getting an instance of the `Camera` object is the first step in the process of directly controlling the camera. As Android's own Camera application does, the recommended way to access the camera is to open `Camera` on a separate thread that's launched from `onCreate()`. This approach is a good idea since it can take a while and might bog down the UI thread. In a

more basic implementation, opening the camera can be deferred to the `onResume()` method to facilitate code reuse and keep the flow of control simple.

Calling `Camera.open()` throws an exception if the camera is already in use by another application, so we wrap it in a `try` block.

```
private boolean safeCameraOpen(int id) {
    boolean qOpened = false;

    try {
        releaseCameraAndPreview();
        mCamera = Camera.open(id);
        qOpened = (mCamera != null);
    } catch (Exception e) {
        Log.e(getString(R.string.app_name), "failed to open Camera");
        e.printStackTrace();
    }

    return qOpened;
}

private void releaseCameraAndPreview() {
    mPreview.setCamera(null);
    if (mCamera != null) {
        mCamera.release();
        mCamera = null;
    }
}
```

Since API level 9, the camera framework supports multiple cameras. If you use the legacy API and call `open()` without an argument, you get the first rear-facing camera.

Create the Camera Preview

Taking a picture usually requires that your users see a preview of their subject before clicking the shutter. To do so, you can use a `SurfaceView` to draw previews of what the camera sensor is picking up.

Preview Class

To get started with displaying a preview, you need preview class. The preview requires an implementation of the `android.view.SurfaceHolder.Callback` interface, which is used to pass image data from the camera hardware to the application.

```
class Preview extends ViewGroup implements SurfaceHolder.Callback {

    SurfaceView mSurfaceView;
    SurfaceHolder mHolder;

    Preview(Context context) {
        super(context);

        mSurfaceView = new SurfaceView(context);
        addView(mSurfaceView);

        // Install a SurfaceHolder.Callback so we get notified when
the
        // underlying surface is created and destroyed.
        mHolder = mSurfaceView.getHolder();
        mHolder.addCallback(this);
        mHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
    }
...
}
```

The preview class must be passed to the `Camera` object before the live image preview can be started, as shown in the next section.

Set and Start the Preview

A camera instance and its related preview must be created in a specific order, with the camera object being first. In the snippet below, the process of initializing the camera is encapsulated so that `Camera.startPreview()` is called by the `setCamera()` method, whenever the user does something to change the camera. The preview must also be restarted in the preview class `surfaceChanged()` callback method.

```
public void setCamera(Camera camera) {
    if (mCamera == camera) { return; }

    stopPreviewAndFreeCamera();

    mCamera = camera;

    if (mCamera != null) {
        List<Size> localSizes =
mCamera.getParameters().getSupportedPreviewSizes();
        mSupportedPreviewSizes = localSizes;
```

```
        requestLayout();

        try {
            mCamera.setPreviewDisplay(mHolder);
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Important: Call startPreview() to start
updating the preview
        // surface. Preview must be started before you
can take a picture.
        mCamera.startPreview();
    }
}
```

Modify Camera Settings

Camera settings change the way that the camera takes pictures, from the zoom level to exposure compensation. This example changes only the preview size; see the source code of the Camera application for many more.

```
public void surfaceChanged(SurfaceHolder holder, int format, int w,
int h) {
    // Now that the size is known, set up the camera parameters and
begin
    // the preview.
    Camera.Parameters parameters = mCamera.getParameters();
    parameters.setPreviewSize(mPreviewSize.width,
mPreviewSize.height);
    requestLayout();
    mCamera.setParameters(parameters);

    // Important: Call startPreview() to start updating the preview
surface.
    // Preview must be started before you can take a picture.
    mCamera.startPreview();
}
```

Set the Preview Orientation

Most camera applications lock the display into landscape mode because that is the natural orientation of the camera sensor. This setting does not prevent you from taking portrait-mode photos, because the orientation of the device is recorded in the EXIF header. The

`setCameraDisplayOrientation()` method lets you change how the preview is displayed without affecting how the image is recorded. However, in Android prior to API level 14, you must stop your preview before changing the orientation and then restart it.

Take a Picture

Use the `Camera.takePicture()` method to take a picture once the preview is started. You can create `Camera.PictureCallback` and `Camera.ShutterCallback` objects and pass them into `Camera.takePicture()`.

If you want to grab images continously, you can create a `Camera.PreviewCallback` that implements `onPreviewFrame()`. For something in between, you can capture only selected preview frames, or set up a delayed action to call `takePicture()`.

Restart the Preview

After a picture is taken, you must restart the preview before the user can take another picture. In this example, the restart is done by overloading the shutter button.

```
@Override
public void onClick(View v) {
    switch(mPreviewState) {
    case K_STATE_FROZEN:
        mCamera.startPreview();
        mPreviewState = K_STATE_PREVIEW;
        break;

    default:
        mCamera.takePicture( null, rawCallback, null);
        mPreviewState = K_STATE_BUSY;
    } // switch
    shutterBtnConfig();
}
```

Stop the Preview and Release the Camera

Once your application is done using the camera, it's time to clean up. In particular, you must release the `Camera` object, or you risk crashing other

applications, including new instances of your own application.

When should you stop the preview and release the camera? Well, having your preview surface destroyed is a pretty good hint that it's time to stop the preview and release the camera, as shown in these methods from the `Preview` class.

```
public void surfaceDestroyed(SurfaceHolder holder) {
    // Surface will be destroyed when we return, so stop the
preview.
    if (mCamera != null) {
        // Call stopPreview() to stop updating the preview surface.
        mCamera.stopPreview();
    }
}

/**
 * When this function returns, mCamera will be null.
 */
private void stopPreviewAndFreeCamera() {

    if (mCamera != null) {
        // Call stopPreview() to stop updating the preview surface.
        mCamera.stopPreview();

        // Important: Call release() to release the camera for use
by other
        // applications. Applications should release the camera
immediately
        // during onPause() and re-open() it during onResume()).
        mCamera.release();

        mCamera = null;
    }
}
```

Earlier in the lesson, this procedure was also part of the `setCamera()` method, so initializing a camera always begins with stopping the preview.

Next Get started Next class

Printing Content

Previous Next Get started

Dependencies and prerequisites

- Android 4.4 (API Level 19) or higher

Video

DevBytes: Android 4.4 Printing API

Android users frequently view content solely on their devices, but there are times when showing someone a screen is not an adequate way to share information. Being able to print information from your Android application gives users a way to see a larger version of the content from your app or share it with another person who is not using your application. Printing also allows them to create a snapshot of information that does not depend on having a device, sufficient battery power, or a wireless network connection.

In Android 4.4 (API level 19) and higher, the framework provides services for printing images and documents directly from Android applications. This training describes how to enable printing in your application, including printing images, HTML pages and creating custom documents for printing.

Lessons

Printing a Photo
This lesson shows you how to print an image.

Printing an HTML Document
This lesson shows you how to print an HTML document.

Printing a Custom Document

This lesson shows you how you connect to the Android print manager, create a print adapter and build content for printing.

Next Get started Next class

Printing Photos

Previous Next Get started

This lesson teaches you to

1. Print an Image

Taking and sharing photos is one of the most popular uses for mobile devices. If your application takes photos, displays them, or allows users to share images, you should consider enabling printing of those images in your application. The Android Support Library provides a convenient function for enabling image printing using a minimal amount of code and simple set of print layout options.

This lesson shows you how to print an image using the v4 support library `PrintHelper` class.

Print an Image

The Android Support Library `PrintHelper` class provides a simple way to print of images. The class has a single layout option, `setScaleMode()`, which allows you to print with one of two options:

- `SCALE_MODE_FIT` - This option sizes the image so that the whole image is shown within the printable area of the page.
- `SCALE_MODE_FILL` - This option scales the image so that it fills the entire printable area of the page. Choosing this setting means that some portion of the top and bottom, or left and right edges of the image is not printed. This option is the default value if you do not set a scale mode.

Both scaling options for `setScaleMode()` keep the existing aspect ratio of the image intact. The following code example shows how to create an

instance of the `PrintHelper` class, set the scaling option, and start the printing process:

```
private void doPhotoPrint() {
    PrintHelper photoPrinter = new PrintHelper(getActivity());
    photoPrinter.setScaleMode(PrintHelper.SCALE_MODE_FIT);
    Bitmap bitmap = BitmapFactory.decodeResource(getResources(),
            R.drawable.droids);
    photoPrinter.printBitmap("droids.jpg - test print", bitmap);
}
```

This method can be called as the action for a menu item. Note that menu items for actions that are not always supported (such as printing) should be placed in the overflow menu. For more information, see the Action Bar design guide.

After the `printBitmap()` method is called, no further action from your application is required. The Android print user interface appears, allowing the user to select a printer and printing options. The user can then print the image or cancel the action. If the user chooses to print the image, a print job is created and a printing notification appears in the system bar.

If you want to include additional content in your printouts beyond just an image, you must construct a print document. For information on creating documents for printing, see the Printing an HTML Document or Printing a Custom Document lessons.

Next Get started Next class

Printing HTML Documents

Previous Next Get started

This lesson teaches you to

1. Load an HTML Document
2. Create a Print Job

Printing out content beyond a simple photo on Android requires composing text and graphics in a print document. The Android framework provides a way to use HTML to compose a document and print it with a minimum of code.

In Android 4.4 (API level 19), the `WebView` class has been updated to enable printing HTML content. The class allows you to load a local HTML resource or download a page from the web, create a print job and hand it off to Android's print services.

This lesson shows you how to quickly build an HTML document containing text and graphics and use `WebView` to print it.

Load an HTML Document

Printing an HTML document with `WebView` involves loading an HTML resource or building an HTML document as a string. This section describes how to build an HTML string and load it into a `WebView` for printing.

This view object is typically used as part of an activity layout. However, if your application is not using a `WebView`, you can create an instance of the class specifically for printing purposes. The main steps for creating this custom print view are:

1. Create a WebViewClient that starts a print job after the HTML resource is loaded.
2. Load the HTML resource into the WebView object.

The following code sample demonstrates how to create a simple WebViewClient and load an HTML document created on the fly:

```
private WebView mWebView;

private void doWebViewPrint() {
    // Create a WebView object specifically for printing
    WebView webView = new WebView(getActivity());
    webView.setWebViewClient(new WebViewClient() {

            public boolean shouldOverrideUrlLoading(WebView view,
String url) {
                return false;
            }

            @Override
            public void onPageFinished(WebView view, String url) {
                Log.i(TAG, "page finished loading " + url);
                createWebPrintJob(view);
                mWebView = null;
            }
    });

    // Generate an HTML document on the fly:
    String htmlDocument = "<html><body><h1>Test
Content</h1><p>Testing, " +
            "testing, testing...</p></body></html>";
    webView.loadDataWithBaseURL(null, htmlDocument, "text/HTML",
"UTF-8", null);

    // Keep a reference to WebView object until you pass the
PrintDocumentAdapter
    // to the PrintManager
    mWebView = webView;
}
```

Note: Make sure your call for generating a print job happens in the onPageFinished() method of the WebViewClient you created in the previous section. If you don't wait until page loading is finished, the print output may be incomplete or blank, or may fail completely.

Note: The example code above holds an instance of the WebView object so that is it not garbage collected before the print job is created. Make sure you do the same in your own implementation, otherwise the print process may fail.

If you want to include graphics in the page, place the graphic files in the `assets/` directory of your project and specify a base URL in the first parameter of the `loadDataWithBaseURL()` method, as shown in the following code example:

```
webView.loadDataWithBaseURL("file:///android_asset/images/",
htmlBody,
        "text/HTML", "UTF-8", null);
```

You can also load a web page for printing by replacing the `loadDataWithBaseURL()` method with `loadUrl()` as shown below.

```
// Print an existing web page (remember to request INTERNET
permission!):
webView.loadUrl("http://developer.android.com/about/index.html");
```

When using `WebView` for creating print documents, you should be aware of the following limitations:

- You cannot add headers or footers, including page numbers, to the document.
- The printing options for the HTML document do not include the ability to print page ranges, for example: Printing page 2 to 4 of a 10 page HTML document is not supported.
- An instance of `WebView` can only process one print job at a time.
- An HTML document containing CSS print attributes, such as landscape properties, is not supported.
- You cannot use JavaScript in a HTML document to trigger printing.

Note: The content of a `WebView` object that is included in a layout can also be printed once it has loaded a document.

If you want to create a more customized print output and have complete control of the content draw on the printed page, jump to the next lesson: Printing a Custom Document lesson.

Create a Print Job

After creating a `WebView` and loading your HTML content, your application is almost done with its part of the printing process. The next steps are

accessing the `PrintManager`, creating a print adapter, and finally, creating a print job. The following example illustrates how to perform these steps:

```
private void createWebPrintJob(WebView webView) {

    // Get a PrintManager instance
    PrintManager printManager = (PrintManager) getActivity()
            .getSystemService(Context.PRINT_SERVICE);

    // Get a print adapter instance
    PrintDocumentAdapter printAdapter =
webView.createPrintDocumentAdapter();

    // Create a print job with name and adapter instance
    String jobName = getString(R.string.app_name) + " Document";
    PrintJob printJob = printManager.print(jobName, printAdapter,
            new PrintAttributes.Builder().build());

    // Save the job object for later status checking
    mPrintJobs.add(printJob);
}
```

This example saves an instance of the `PrintJob` object for use by the application, which is not required. Your application may use this object to track the progress of the print job as it's being processed. This approach is useful when you want to monitor the status of the print job in you application for completion, failure, or user cancellation. Creating an in-app notification is not required, because the print framework automatically creates a system notification for the print job.

Next Get started Next class

Printing Custom Documents

Previous Next Get started

This lesson teaches you to

1. Connect to the Print Manager
2. Create a Print Adapter
    1. Compute print document info
    2. Write a print document file
3. Drawing PDF Page Content

For some applications, such as drawing apps, page layout apps and other apps that focus on graphic output, creating beautiful printed pages is a key feature. In this case, it is not enough to print an image or an HTML document. The print output for these types of applications requires precise control of everything that goes into a page, including fonts, text flow, page breaks, headers, footers, and graphic elements.

Creating print output that is completely customized for your application requires more programming investment than the previously discussed approaches. You must build components that communicate with the print framework, adjust to printer settings, draw page elements and manage printing on multiple pages.

This lesson shows you how you connect with the print manager, create a print adapter and build content for printing.

Connect to the Print Manager

When your application manages the printing process directly, the first step after receiving a print request from your user is to connect to the Android print framework and obtain an instance of the `PrintManager`

class. This class allows you to initialize a print job and begin the printing lifecycle. The following code example shows how to get the print manager and start the printing process.

```
private void doPrint() {
    // Get a PrintManager instance
    PrintManager printManager = (PrintManager) getActivity()
            .getSystemService(Context.PRINT_SERVICE);

    // Set job name, which will be displayed in the print queue
    String jobName = getActivity().getString(R.string.app_name) + "
Document";

    // Start a print job, passing in a PrintDocumentAdapter
implementation
    // to handle the generation of a print document
    printManager.print(jobName, new
MyPrintDocumentAdapter(getActivity()),
            null); //
}
```

The example code above demonstrates how to name a print job and set an instance of the `PrintDocumentAdapter` class which handles the steps of the printing lifecycle. The implementation of the print adapter class is discussed in the next section.

Note: The last parameter in the `print()` method takes a `PrintAttributes` object. You can use this parameter to provide hints to the printing framework and pre-set options based on the previous printing cycle, thereby improving the user experience. You may also use this parameter to set options that are more appropriate to the content being printed, such as setting the orientation to landscape when printing a photo that is in that orientation.

Create a Print Adapter

A print adapter interacts with the Android print framework and handles the steps of the printing process. This process requires users to select printers and print options before creating a document for printing. These selections can influence the final output as the user chooses printers with different output capabilities, different page sizes, or different page orientations. As these selections are made, the print framework asks your adapter to lay out and generate a print document, in preparation for final

output. Once a user taps the print button, the framework takes the final print document and passes it to a print provider for output. During the printing process, users can choose to cancel the print action, so your print adapter must also listen for and react to a cancellation requests.

The `PrintDocumentAdapter` abstract class is designed to handle the printing lifecycle, which has four main callback methods. You must implement these methods in your print adapter in order to interact properly with the print framework:

- `onStart()` - Called once at the beginning of the print process. If your application has any one-time preparation tasks to perform, such as getting a snapshot of the data to be printed, execute them here. Implementing this method in your adapter is not required.
- `onLayout()` - Called each time a user changes a print setting which impacts the output, such as a different page size, or page orientation, giving your application an opportunity to compute the layout of the pages to be printed. At the minimum, this method must return how many pages are expected in the printed document.
- `onWrite()` - Called to render printed pages into a file to be printed. This method may be called one or more times after each `onLayout()` call.
- `onFinish()` - Called once at the end of the print process. If your application has any one-time tear-down tasks to perform, execute them here. Implementing this method in your adapter is not required.

The following sections describe how to implement the layout and write methods, which are critical to the functioning of a print adapter.

Note: These adapter methods are called on the main thread of your application. If you expect the execution of these methods in your implementation to take a significant amount of time, implement them to execute within a separate thread. For example, you can encapsulate the layout or print document writing work in separate `AsyncTask` objects.

Compute print document info

Within an implementation of the PrintDocumentAdapter class, your application must be able to specify the type of document it is creating and calculate the total number of pages for print job, given information about the printed page size. The implementation of the onLayout() method in the adapter makes these calculations and provides information about the expected output of the print job in a PrintDocumentInfo class, including the number of pages and content type. The following code example shows a basic implementation of the onLayout() method for a PrintDocumentAdapter:

```
@Override
public void onLayout(PrintAttributes oldAttributes,
                     PrintAttributes newAttributes,
                     CancellationSignal cancellationSignal,
                     LayoutResultCallback callback,
                     Bundle metadata) {
    // Create a new PdfDocument with the requested page attributes
    mPdfDocument = new PrintedPdfDocument(getActivity(),
newAttributes);

    // Respond to cancellation request
    if (cancellationSignal.isCancelled() ) {
        callback.onLayoutCancelled();
        return;
    }

    // Compute the expected number of printed pages
    int pages = computePageCount(newAttributes);

    if (pages > 0) {
        // Return print information to print framework
        PrintDocumentInfo info = new PrintDocumentInfo
                .Builder("print_output.pdf")

.setContentType(PrintDocumentInfo.CONTENT_TYPE_DOCUMENT)
                .setPageCount(pages);
                .build();
        // Content layout reflow is complete
        callback.onLayoutFinished(info, true);
    } else {
        // Otherwise report an error to the print framework
        callback.onLayoutFailed("Page count calculation failed.");
    }
}
```

The execution of onLayout() method can have three outcomes: completion, cancellation, or failure in the case where calculation of the layout cannot be completed. You must indicate one of these results by calling the appropriate method of the PrintDocumentAdapter.LayoutResultCallback object.

Note: The boolean parameter of the `onLayoutFinished()` method indicates whether or not the layout content has actually changed since the last request. Setting this parameter properly allows the print framework to avoid unnecessarily calling the `onWrite()` method, essentially caching the previously written print document and improving performance.

The main work of `onLayout()` is calculating the number of pages that are expected as output given the attributes of the printer. How you calculate this number is highly dependent on how your application lays out pages for printing. The following code example shows an implementation where the number of pages is determined by the print orientation:

```
private int computePageCount(PrintAttributes printAttributes) {
    int itemsPerPage = 4; // default item count for portrait mode

    MediaSize pageSize = printAttributes.getMediaSize();
    if (!pageSize.isPortrait()) {
        // Six items per page in landscape orientation
        itemsPerPage = 6;
    }

    // Determine number of print items
    int printItemCount = getPrintItemCount();

    return (int) Math.ceil(printItemCount / itemsPerPage);
}
```

Write a print document file

When it is time to write print output to a file, the Android print framework calls the `onWrite()` method of your application's `PrintDocumentAdapter` class. The method's parameters specify which pages should be written and the output file to be used. Your implementation of this method must then render each requested page of content to a multi-page PDF document file. When this process is complete, you call the `onWriteFinished()` method of the callback object.

Note: The Android print framework may call the `onWrite()` method one or more times for every call to `onLayout()`. For this reason, it is important to set the boolean parameter of `onLayoutFinished()` method to `false` when the print content layout has not changed, to avoid unnecessary re-writes of the print document.

Note: The boolean parameter of the `onLayoutFinished()` method indicates whether or not the layout content has actually changed since the last request. Setting this parameter properly allows the print framework to avoid unnecessarily calling the `onLayout()` method, essentially caching the previously written print document and improving performance.

The following sample demonstrates the basic mechanics of this process using the `PrintedPdfDocument` class to create a PDF file:

```
@Override
public void onWrite(final PageRange[] pageRanges,
                    final ParcelFileDescriptor destination,
                    final CancellationSignal cancellationSignal,
                    final WriteResultCallback callback) {
    // Iterate over each page of the document,
    // check if it's in the output range.
    for (int i = 0; i < totalPages; i++) {
        // Check to see if this page is in the output range.
        if (containsPage(pageRanges, i)) {
            // If so, add it to writtenPagesArray.
writtenPagesArray.size()
            // is used to compute the next output page index.
            writtenPagesArray.append(writtenPagesArray.size(), i);
            PdfDocument.Page page = mPdfDocument.startPage(i);

            // check for cancellation
            if (cancellationSignal.isCancelled()) {
                callback.onWriteCancelled();
                mPdfDocument.close();
                mPdfDocument = null;
                return;
            }

            // Draw page content for printing
            drawPage(page);

            // Rendering is complete, so page can be finalized.
            mPdfDocument.finishPage(page);
        }
    }

    // Write PDF document to file
    try {
        mPdfDocument.writeTo(new FileOutputStream(
                destination.getFileDescriptor()));
    } catch (IOException e) {
        callback.onWriteFailed(e.toString());
        return;
    } finally {
        mPdfDocument.close();
        mPdfDocument = null;
    }
    PageRange[] writtenPages = computeWrittenPages();
    // Signal the print framework the document is complete
```

```
    callback.onWriteFinished(writtenPages);

    ...
}
```

This sample delegates rendering of PDF page content to `drawPage()` method, which is discussed in the next section.

As with layout, execution of `onWrite()` method can have three outcomes: completion, cancellation, or failure in the case where the the content cannot be written. You must indicate one of these results by calling the appropriate method of the `PrintDocumentAdapter.WriteResultCallback` object.

Note: Rendering a document for printing can be a resource-intensive operation. In order to avoid blocking the main user interface thread of your application, you should consider performing the page rendering and writing operations on a separate thread, for example in an `AsyncTask`. For more information about working with execution threads like asynchronous tasks, see Processes and Threads.

Drawing PDF Page Content

When your application prints, your application must generate a PDF document and pass it to the Android print framework for printing. You can use any PDF generation library for this purpose. This lesson shows how to use the `PrintedPdfDocument` class to generate PDF pages from your content.

The `PrintedPdfDocument` class uses a `Canvas` object to draw elements on an PDF page, similar to drawing on an activity layout. You can draw elements on the printed page using the `Canvas` draw methods. The following example code demonstrates how to draw some simple elements on a PDF document page using these methods:

```
private void drawPage(PdfDocument.Page page) {
    Canvas canvas = page.getCanvas();

    // units are in points (1/72 of an inch)
    int titleBaseLine = 72;
    int leftMargin = 54;
```

```
    Paint paint = new Paint();
    paint.setColor(Color.BLACK);
    paint.setTextSize(36);
    canvas.drawText("Test Title", leftMargin, titleBaseLine, paint);

    paint.setTextSize(11);
    canvas.drawText("Test paragraph", leftMargin, titleBaseLine +
25, paint);

    paint.setColor(Color.BLUE);
    canvas.drawRect(100, 100, 172, 172, paint);
}
```

When using `Canvas` to draw on a PDF page, elements are specified in points, which is 1/72 of an inch. Make sure you use this unit of measure for specifying the size of elements on the page. For positioning of drawn elements, the coordinate system starts at 0,0 for the top left corner of the page.

Tip: While the `Canvas` object allows you to place print elements on the edge of a PDF document, many printers are not able to print to the edge of a physical piece of paper. Make sure that you account for the unprintable edges of the page when you build a print document with this class.

Next Get started Next class

Building Apps with Graphics & Animation

These classes teach you how to accomplish tasks with graphics that can give your app an edge on the competition. If you want to go beyond the basic user interface to create a beautiful visual experience, these classes will help you get there.

Displaying Bitmaps Efficiently

Previous Next Get started

Dependencies and prerequisites

- Android 2.1 (API Level 7) or higher
- Support Library

Try it out

Download the sample

DisplayingBitmaps.zip

Video

DevBytes: Bitmap Allocation

Video

DevBytes: Making Apps Beautiful - Part 4 - Performance Tuning

Learn how to use common techniques to process and load `Bitmap` objects in a way that keeps your user interface (UI) components responsive and avoids exceeding your application memory limit. If you're not careful, bitmaps can quickly consume your available memory budget leading to an application crash due to the dreaded exception:

```
java.lang.OutofMemoryError: bitmap size exceeds VM budget.
```

There are a number of reasons why loading bitmaps in your Android application is tricky:

- Mobile devices typically have constrained system resources. Android

devices can have as little as 16MB of memory available to a single application. The Android Compatibility Definition Document (CDD), *Section 3.7. Virtual Machine Compatibility* gives the required minimum application memory for various screen sizes and densities. Applications should be optimized to perform under this minimum memory limit. However, keep in mind many devices are configured with higher limits.

- Bitmaps take up a lot of memory, especially for rich images like photographs. For example, the camera on the Galaxy Nexus takes photos up to 2592x1936 pixels (5 megapixels). If the bitmap configuration used is `ARGB_8888` (the default from the Android 2.3 onward) then loading this image into memory takes about 19MB of memory (2592*1936*4 bytes), immediately exhausting the per-app limit on some devices.
- Android app UI's frequently require several bitmaps to be loaded at once. Components such as `ListView`, `GridView` and `ViewPager` commonly include multiple bitmaps on-screen at once with many more potentially off-screen ready to show at the flick of a finger.

Lessons

Loading Large Bitmaps Efficiently

This lesson walks you through decoding large bitmaps without exceeding the per application memory limit.

Processing Bitmaps Off the UI Thread

Bitmap processing (resizing, downloading from a remote source, etc.) should never take place on the main UI thread. This lesson walks you through processing bitmaps in a background thread using `AsyncTask` and explains how to handle concurrency issues.

Caching Bitmaps

This lesson walks you through using a memory and disk bitmap cache to improve the responsiveness and fluidity of your UI when loading multiple bitmaps.

Managing Bitmap Memory

This lesson explains how to manage bitmap memory to maximize your app's performance.

Displaying Bitmaps in Your UI

This lesson brings everything together, showing you how to load multiple bitmaps into components like `ViewPager` and `GridView` using a background thread and bitmap cache.

Next Get started Next class

Loading Large Bitmaps Efficiently

Previous Next Get started

This lesson teaches you to

1. Read Bitmap Dimensions and Type
2. Load a Scaled Down Version into Memory

Try it out

Download the sample

DisplayingBitmaps.zip

Images come in all shapes and sizes. In many cases they are larger than required for a typical application user interface (UI). For example, the system Gallery application displays photos taken using your Android devices's camera which are typically much higher resolution than the screen density of your device.

Given that you are working with limited memory, ideally you only want to load a lower resolution version in memory. The lower resolution version should match the size of the UI component that displays it. An image with a higher resolution does not provide any visible benefit, but still takes up precious memory and incurs additional performance overhead due to additional on the fly scaling.

This lesson walks you through decoding large bitmaps without exceeding the per application memory limit by loading a smaller subsampled version in memory.

Read Bitmap Dimensions and Type

The `BitmapFactory` class provides several decoding methods (`decodeByteArray()`, `decodeFile()`, `decodeResource()`, etc.) for creating a `Bitmap` from various sources. Choose the most appropriate decode method based on your image data source. These methods attempt to allocate memory for the constructed bitmap and therefore can easily result in an `OutOfMemory` exception. Each type of decode method has additional signatures that let you specify decoding options via the `BitmapFactory.Options` class. Setting the `inJustDecodeBounds` property to `true` while decoding avoids memory allocation, returning `null` for the bitmap object but setting `outWidth`, `outHeight` and `outMimeType`. This technique allows you to read the dimensions and type of the image data prior to construction (and memory allocation) of the bitmap.

```
BitmapFactory.Options options = new BitmapFactory.Options();
options.inJustDecodeBounds = true;
BitmapFactory.decodeResource(getResources(), R.id.myimage, options);
int imageHeight = options.outHeight;
int imageWidth = options.outWidth;
String imageType = options.outMimeType;
```

To avoid `java.lang.OutOfMemory` exceptions, check the dimensions of a bitmap before decoding it, unless you absolutely trust the source to provide you with predictably sized image data that comfortably fits within the available memory.

Load a Scaled Down Version into Memory

Now that the image dimensions are known, they can be used to decide if the full image should be loaded into memory or if a subsampled version should be loaded instead. Here are some factors to consider:

- Estimated memory usage of loading the full image in memory.
- Amount of memory you are willing to commit to loading this image given any other memory requirements of your application.
- Dimensions of the target `ImageView` or UI component that the image is to be loaded into.
- Screen size and density of the current device.

For example, it’s not worth loading a 1024x768 pixel image into memory

if it will eventually be displayed in a 128x96 pixel thumbnail in an `ImageView`.

To tell the decoder to subsample the image, loading a smaller version into memory, set `inSampleSize` to `true` in your `BitmapFactory.Options` object. For example, an image with resolution 2048x1536 that is decoded with an `inSampleSize` of 4 produces a bitmap of approximately 512x384. Loading this into memory uses 0.75MB rather than 12MB for the full image (assuming a bitmap configuration of `ARGB_8888`). Here's a method to calculate a sample size value that is a power of two based on a target width and height:

```
public static int calculateInSampleSize(
            BitmapFactory.Options options, int reqWidth, int
reqHeight) {
    // Raw height and width of image
    final int height = options.outHeight;
    final int width = options.outWidth;
    int inSampleSize = 1;

    if (height > reqHeight || width > reqWidth) {

        final int halfHeight = height / 2;
        final int halfWidth = width / 2;

        // Calculate the largest inSampleSize value that is a power
of 2 and keeps both
        // height and width larger than the requested height and
width.
        while ((halfHeight / inSampleSize) > reqHeight
                && (halfWidth / inSampleSize) > reqWidth) {
            inSampleSize *= 2;
        }
    }

    return inSampleSize;
}
```

Note: A power of two value is calculated because the decoder uses a final value by rounding down to the nearest power of two, as per the `inSampleSize` documentation.

To use this method, first decode with `inJustDecodeBounds` set to `true`, pass the options through and then decode again using the new `inSampleSize` value and `inJustDecodeBounds` set to `false`:

```
public static Bitmap decodeSampledBitmapFromResource(Resources res,
int resId,
        int reqWidth, int reqHeight) {

    // First decode with inJustDecodeBounds=true to check dimensions
    final BitmapFactory.Options options = new
BitmapFactory.Options();
    options.inJustDecodeBounds = true;
    BitmapFactory.decodeResource(res, resId, options);

    // Calculate inSampleSize
    options.inSampleSize = calculateInSampleSize(options, reqWidth,
reqHeight);

    // Decode bitmap with inSampleSize set
    options.inJustDecodeBounds = false;
    return BitmapFactory.decodeResource(res, resId, options);
}
```

This method makes it easy to load a bitmap of arbitrarily large size into an `ImageView` that displays a 100x100 pixel thumbnail, as shown in the following example code:

```
mImageView.setImageBitmap(
    decodeSampledBitmapFromResource(getResources(), R.id.myimage,
100, 100));
```

You can follow a similar process to decode bitmaps from other sources, by substituting the appropriate `BitmapFactory.decode*` method as needed.

Next Get started Next class

Processing Bitmaps Off the UI Thread

Previous Next Get started

This lesson teaches you to

1. Use an AsyncTask
2. Handle Concurrency

You should also read

- Designing for Responsiveness
- Multithreading for Performance

Try it out

Download the sample

DisplayingBitmaps.zip

The `BitmapFactory.decode*` methods, discussed in the Load Large Bitmaps Efficiently lesson, should not be executed on the main UI thread if the source data is read from disk or a network location (or really any source other than memory). The time this data takes to load is unpredictable and depends on a variety of factors (speed of reading from disk or network, size of image, power of CPU, etc.). If one of these tasks blocks the UI thread, the system flags your application as non-responsive and the user has the option of closing it (see Designing for Responsiveness for more information).

This lesson walks you through processing bitmaps in a background

thread using AsyncTask and shows you how to handle concurrency issues.

Use an AsyncTask

The AsyncTask class provides an easy way to execute some work in a background thread and publish the results back on the UI thread. To use it, create a subclass and override the provided methods. Here's an example of loading a large image into an ImageView using AsyncTask and decodeSampledBitmapFromResource():

```
class BitmapWorkerTask extends AsyncTask<Integer, Void, Bitmap> {
    private final WeakReference<ImageView> imageViewReference;
    private int data = 0;

    public BitmapWorkerTask(ImageView imageView) {
        // Use a WeakReference to ensure the ImageView can be
garbage collected
        imageViewReference = new WeakReference<ImageView>(imageView);
    }

    // Decode image in background.
    @Override
    protected Bitmap doInBackground(Integer... params) {
        data = params[0];
        return decodeSampledBitmapFromResource(getResources(), data,
100, 100));
    }

    // Once complete, see if ImageView is still around and set
bitmap.
    @Override
    protected void onPostExecute(Bitmap bitmap) {
        if (imageViewReference != null && bitmap != null) {
            final ImageView imageView = imageViewReference.get();
            if (imageView != null) {
                imageView.setImageBitmap(bitmap);
            }
        }
    }
}
```

The WeakReference to the ImageView ensures that the AsyncTask does not prevent the ImageView and anything it references from being garbage collected. There's no guarantee the ImageView is still around when the task finishes, so you must also check the reference in onPostExecute(). The ImageView may no longer exist, if for example, the user navigates away from the activity or if a configuration change happens before the

task finishes.

To start loading the bitmap asynchronously, simply create a new task and execute it:

```
public void loadBitmap(int resId, ImageView imageView) {
    BitmapWorkerTask task = new BitmapWorkerTask(imageView);
    task.execute(resId);
}
```

Handle Concurrency

Common view components such as `ListView` and `GridView` introduce another issue when used in conjunction with the `AsyncTask` as demonstrated in the previous section. In order to be efficient with memory, these components recycle child views as the user scrolls. If each child view triggers an `AsyncTask`, there is no guarantee that when it completes, the associated view has not already been recycled for use in another child view. Furthermore, there is no guarantee that the order in which asynchronous tasks are started is the order that they complete.

The blog post Multithreading for Performance further discusses dealing with concurrency, and offers a solution where the `ImageView` stores a reference to the most recent `AsyncTask` which can later be checked when the task completes. Using a similar method, the `AsyncTask` from the previous section can be extended to follow a similar pattern.

Create a dedicated `Drawable` subclass to store a reference back to the worker task. In this case, a `BitmapDrawable` is used so that a placeholder image can be displayed in the `ImageView` while the task completes:

```
static class AsyncDrawable extends BitmapDrawable {
    private final WeakReference<BitmapWorkerTask>
bitmapWorkerTaskReference;

    public AsyncDrawable(Resources res, Bitmap bitmap,
            BitmapWorkerTask bitmapWorkerTask) {
        super(res, bitmap);
        bitmapWorkerTaskReference =
            new WeakReference<BitmapWorkerTask>(bitmapWorkerTask);
    }
```

```
    public BitmapWorkerTask getBitmapWorkerTask() {
        return bitmapWorkerTaskReference.get();
    }
}
```

Before executing the `BitmapWorkerTask`, you create an `AsyncDrawable` and bind it to the target `ImageView`:

```
public void loadBitmap(int resId, ImageView imageView) {
    if (cancelPotentialWork(resId, imageView)) {
        final BitmapWorkerTask task = new
BitmapWorkerTask(imageView);
        final AsyncDrawable asyncDrawable =
                new AsyncDrawable(getResources(), mPlaceHolderBitmap,
task);
        imageView.setImageDrawable(asyncDrawable);
        task.execute(resId);
    }
}
```

The `cancelPotentialWork` method referenced in the code sample above checks if another running task is already associated with the `ImageView`. If so, it attempts to cancel the previous task by calling `cancel()`. In a small number of cases, the new task data matches the existing task and nothing further needs to happen. Here is the implementation of `cancelPotentialWork`:

```
public static boolean cancelPotentialWork(int data, ImageView
imageView) {
    final BitmapWorkerTask bitmapWorkerTask =
getBitmapWorkerTask(imageView);

    if (bitmapWorkerTask != null) {
        final int bitmapData = bitmapWorkerTask.data;
        // If bitmapData is not yet set or it differs from the new
data
        if (bitmapData == 0 || bitmapData != data) {
            // Cancel previous task
            bitmapWorkerTask.cancel(true);
        } else {
            // The same work is already in progress
            return false;
        }
    }
    // No task associated with the ImageView, or an existing task
was cancelled
    return true;
}
```

A helper method, `getBitmapWorkerTask()`, is used above to retrieve the task associated with a particular `ImageView`:

```
private static BitmapWorkerTask getBitmapWorkerTask(ImageView
```

```
imageView) {
   if (imageView != null) {
       final Drawable drawable = imageView.getDrawable();
       if (drawable instanceof AsyncDrawable) {
           final AsyncDrawable asyncDrawable = (AsyncDrawable)
drawable;
           return asyncDrawable.getBitmapWorkerTask();
       }
    }
    return null;
}
```

The last step is updating `onPostExecute()` in `BitmapWorkerTask` so that it checks if the task is cancelled and if the current task matches the one associated with the `ImageView`:

```
class BitmapWorkerTask extends AsyncTask<Integer, Void, Bitmap> {
    ...

    @Override
    protected void onPostExecute(Bitmap bitmap) {
        if (isCancelled()) {
            bitmap = null;
        }

        if (imageViewReference != null && bitmap != null) {
            final ImageView imageView = imageViewReference.get();
            final BitmapWorkerTask bitmapWorkerTask =
                    getBitmapWorkerTask(imageView);
            if (this == bitmapWorkerTask && imageView != null) {
                imageView.setImageBitmap(bitmap);
            }
        }
    }
}
```

This implementation is now suitable for use in `ListView` and `GridView` components as well as any other components that recycle their child views. Simply call `loadBitmap` where you normally set an image to your `ImageView`. For example, in a `GridView` implementation this would be in the `getView()` method of the backing adapter.

Next Get started Next class

Caching Bitmaps

Previous Next Get started

This lesson teaches you to

1. Use a Memory Cache
2. Use a Disk Cache
3. Handle Configuration Changes

You should also read

- Handling Runtime Changes

Try it out

Download the sample

DisplayingBitmaps.zip

Loading a single bitmap into your user interface (UI) is straightforward, however things get more complicated if you need to load a larger set of images at once. In many cases (such as with components like `ListView`, `GridView` or `ViewPager`), the total number of images on-screen combined with images that might soon scroll onto the screen are essentially unlimited.

Memory usage is kept down with components like this by recycling the child views as they move off-screen. The garbage collector also frees up your loaded bitmaps, assuming you don't keep any long lived references. This is all good and well, but in order to keep a fluid and fast-loading UI you want to avoid continually processing these images each time they come back on-screen. A memory and disk cache can often help here,

allowing components to quickly reload processed images.

This lesson walks you through using a memory and disk bitmap cache to improve the responsiveness and fluidity of your UI when loading multiple bitmaps.

Use a Memory Cache

A memory cache offers fast access to bitmaps at the cost of taking up valuable application memory. The `LruCache` class (also available in the Support Library for use back to API Level 4) is particularly well suited to the task of caching bitmaps, keeping recently referenced objects in a strong referenced `LinkedHashMap` and evicting the least recently used member before the cache exceeds its designated size.

Note: In the past, a popular memory cache implementation was a `SoftReference` or `WeakReference` bitmap cache, however this is not recommended. Starting from Android 2.3 (API Level 9) the garbage collector is more aggressive with collecting soft/weak references which makes them fairly ineffective. In addition, prior to Android 3.0 (API Level 11), the backing data of a bitmap was stored in native memory which is not released in a predictable manner, potentially causing an application to briefly exceed its memory limits and crash.

In order to choose a suitable size for a `LruCache`, a number of factors should be taken into consideration, for example:

- How memory intensive is the rest of your activity and/or application?
- How many images will be on-screen at once? How many need to be available ready to come on-screen?
- What is the screen size and density of the device? An extra high density screen (xhdpi) device like Galaxy Nexus will need a larger cache to hold the same number of images in memory compared to a device like Nexus S (hdpi).
- What dimensions and configuration are the bitmaps and therefore how much memory will each take up?
- How frequently will the images be accessed? Will some be accessed

more frequently than others? If so, perhaps you may want to keep certain items always in memory or even have multiple `LruCache` objects for different groups of bitmaps.
- Can you balance quality against quantity? Sometimes it can be more useful to store a larger number of lower quality bitmaps, potentially loading a higher quality version in another background task.

There is no specific size or formula that suits all applications, it's up to you to analyze your usage and come up with a suitable solution. A cache that is too small causes additional overhead with no benefit, a cache that is too large can once again cause `java.lang.OutOfMemory` exceptions and leave the rest of your app little memory to work with.

Here's an example of setting up a `LruCache` for bitmaps:

```
private LruCache<String, Bitmap> mMemoryCache;

@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    // Get max available VM memory, exceeding this amount will throw
an
    // OutOfMemory exception. Stored in kilobytes as LruCache takes
an
    // int in its constructor.
    final int maxMemory = (int) (Runtime.getRuntime().maxMemory() /
1024);

    // Use 1/8th of the available memory for this memory cache.
    final int cacheSize = maxMemory / 8;

    mMemoryCache = new LruCache<String, Bitmap>(cacheSize) {
        @Override
        protected int sizeOf(String key, Bitmap bitmap) {
            // The cache size will be measured in kilobytes rather
than
            // number of items.
            return bitmap.getByteCount() / 1024;
        }
    };
    ...
}

public void addBitmapToMemoryCache(String key, Bitmap bitmap) {
    if (getBitmapFromMemCache(key) == null) {
        mMemoryCache.put(key, bitmap);
    }
}

public Bitmap getBitmapFromMemCache(String key) {
    return mMemoryCache.get(key);
}
```

Note: In this example, one eighth of the application memory is allocated for our cache. On a normal/hdpi device this is a minimum of around 4MB (32/8). A full screen `GridView` filled with images on a device with 800x480 resolution would use around 1.5MB (800*480*4 bytes), so this would cache a minimum of around 2.5 pages of images in memory.

When loading a bitmap into an `ImageView`, the `LruCache` is checked first. If an entry is found, it is used immediately to update the `ImageView`, otherwise a background thread is spawned to process the image:

```
public void loadBitmap(int resId, ImageView imageView) {
    final String imageKey = String.valueOf(resId);

    final Bitmap bitmap = getBitmapFromMemCache(imageKey);
    if (bitmap != null) {
        mImageView.setImageBitmap(bitmap);
    } else {
        mImageView.setImageResource(R.drawable.image_placeholder);
        BitmapWorkerTask task = new BitmapWorkerTask(mImageView);
        task.execute(resId);
    }
}
```

The `BitmapWorkerTask` also needs to be updated to add entries to the memory cache:

```
class BitmapWorkerTask extends AsyncTask<Integer, Void, Bitmap> {
    ...
    // Decode image in background.
    @Override
    protected Bitmap doInBackground(Integer... params) {
        final Bitmap bitmap = decodeSampledBitmapFromResource(
                getResources(), params[0], 100, 100));
        addBitmapToMemoryCache(String.valueOf(params[0]), bitmap);
        return bitmap;
    }
    ...
}
```

Use a Disk Cache

A memory cache is useful in speeding up access to recently viewed bitmaps, however you cannot rely on images being available in this cache. Components like `GridView` with larger datasets can easily fill up a memory cache. Your application could be interrupted by another task like a phone call, and while in the background it might be killed and the

memory cache destroyed. Once the user resumes, your application has to process each image again.

A disk cache can be used in these cases to persist processed bitmaps and help decrease loading times where images are no longer available in a memory cache. Of course, fetching images from disk is slower than loading from memory and should be done in a background thread, as disk read times can be unpredictable.

Note: A `ContentProvider` might be a more appropriate place to store cached images if they are accessed more frequently, for example in an image gallery application.

The sample code of this class uses a `DiskLruCache` implementation that is pulled from the Android source. Here's updated example code that adds a disk cache in addition to the existing memory cache:

```
private DiskLruCache mDiskLruCache;
private final Object mDiskCacheLock = new Object();
private boolean mDiskCacheStarting = true;
private static final int DISK_CACHE_SIZE = 1024 * 1024 * 10; // 10MB
private static final String DISK_CACHE_SUBDIR = "thumbnails";

@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    // Initialize memory cache
    ...
    // Initialize disk cache on background thread
    File cacheDir = getDiskCacheDir(this, DISK_CACHE_SUBDIR);
    new InitDiskCacheTask().execute(cacheDir);
    ...
}

class InitDiskCacheTask extends AsyncTask<File, Void, Void> {
    @Override
    protected Void doInBackground(File... params) {
        synchronized (mDiskCacheLock) {
            File cacheDir = params[0];
            mDiskLruCache = DiskLruCache.open(cacheDir,
DISK_CACHE_SIZE);
            mDiskCacheStarting = false; // Finished initialization
            mDiskCacheLock.notifyAll(); // Wake any waiting threads
        }
        return null;
    }
}

class BitmapWorkerTask extends AsyncTask<Integer, Void, Bitmap> {
    ...
    // Decode image in background.
```

```
    @Override
    protected Bitmap doInBackground(Integer... params) {
        final String imageKey = String.valueOf(params[0]);

        // Check disk cache in background thread
        Bitmap bitmap = getBitmapFromDiskCache(imageKey);

        if (bitmap == null) { // Not found in disk cache
            // Process as normal
            final Bitmap bitmap = decodeSampledBitmapFromResource(
                    getResources(), params[0], 100, 100));
        }

        // Add final bitmap to caches
        addBitmapToCache(imageKey, bitmap);

        return bitmap;
    }
    ...
}

public void addBitmapToCache(String key, Bitmap bitmap) {
    // Add to memory cache as before
    if (getBitmapFromMemCache(key) == null) {
        mMemoryCache.put(key, bitmap);
    }

    // Also add to disk cache
    synchronized (mDiskCacheLock) {
        if (mDiskLruCache != null && mDiskLruCache.get(key) == null)
{
            mDiskLruCache.put(key, bitmap);
        }
    }
}

public Bitmap getBitmapFromDiskCache(String key) {
    synchronized (mDiskCacheLock) {
        // Wait while disk cache is started from background thread
        while (mDiskCacheStarting) {
            try {
                mDiskCacheLock.wait();
            } catch (InterruptedException e) {}
        }
        if (mDiskLruCache != null) {
            return mDiskLruCache.get(key);
        }
    }
    return null;
}

// Creates a unique subdirectory of the designated app cache
directory. Tries to use external
// but if not mounted, falls back on internal storage.
public static File getDiskCacheDir(Context context, String
uniqueName) {
    // Check if media is mounted or storage is built-in, if so, try
and use external cache dir
    // otherwise use internal cache dir
    final String cachePath =
```

```
Environment.MEDIA_MOUNTED.equals(Environment.getExternalStorageState())
||
                    !isExternalStorageRemovable() ?
getExternalCacheDir(context).getPath() :
                            context.getCacheDir().getPath();

    return new File(cachePath + File.separator + uniqueName);
}
```

Note: Even initializing the disk cache requires disk operations and therefore should not take place on the main thread. However, this does mean there's a chance the cache is accessed before initialization. To address this, in the above implementation, a lock object ensures that the app does not read from the disk cache until the cache has been initialized.

While the memory cache is checked in the UI thread, the disk cache is checked in the background thread. Disk operations should never take place on the UI thread. When image processing is complete, the final bitmap is added to both the memory and disk cache for future use.

Handle Configuration Changes

Runtime configuration changes, such as a screen orientation change, cause Android to destroy and restart the running activity with the new configuration (For more information about this behavior, see Handling Runtime Changes). You want to avoid having to process all your images again so the user has a smooth and fast experience when a configuration change occurs.

Luckily, you have a nice memory cache of bitmaps that you built in the Use a Memory Cache section. This cache can be passed through to the new activity instance using a `Fragment` which is preserved by calling `setRetainInstance(true)`). After the activity has been recreated, this retained `Fragment` is reattached and you gain access to the existing cache object, allowing images to be quickly fetched and re-populated into the `ImageView` objects.

Here’s an example of retaining a `LruCache` object across configuration changes using a `Fragment`:

```
private LruCache<String, Bitmap> mMemoryCache;

@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    RetainFragment retainFragment =

RetainFragment.findOrCreateRetainFragment(getFragmentManager());
    mMemoryCache = retainFragment.mRetainedCache;
    if (mMemoryCache == null) {
        mMemoryCache = new LruCache<String, Bitmap>(cacheSize) {
            ... // Initialize cache here as usual
        }
        retainFragment.mRetainedCache = mMemoryCache;
    }
    ...
}

class RetainFragment extends Fragment {
    private static final String TAG = "RetainFragment";
    public LruCache<String, Bitmap> mRetainedCache;

    public RetainFragment() {}

    public static RetainFragment
findOrCreateRetainFragment(FragmentManager fm) {
        RetainFragment fragment = (RetainFragment)
fm.findFragmentByTag(TAG);
        if (fragment == null) {
            fragment = new RetainFragment();
            fm.beginTransaction().add(fragment, TAG).commit();
        }
        return fragment;
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setRetainInstance(true);
    }
}
```

To test this out, try rotating a device both with and without retaining the `Fragment`. You should notice little to no lag as the images populate the activity almost instantly from memory when you retain the cache. Any images not found in the memory cache are hopefully available in the disk cache, if not, they are processed as usual.

Next Get started Next class

Managing Bitmap Memory

Previous Next Get started

This lesson teaches you to

1. Manage Memory on Android 2.3.3 and Lower
2. Manage Memory on Android 3.0 and Higher

You should also read

- Memory Analysis for Android Applications blog post
- Memory management for Android Apps Google I/O presentation
- Android Design: Swipe Views
- Android Design: Grid Lists

Try it out

Download the sample

DisplayingBitmaps.zip

In addition to the steps described in Caching Bitmaps, there are specific things you can do to facilitate garbage collection and bitmap reuse. The recommended strategy depends on which version(s) of Android you are targeting. The `BitmapFun` sample app included with this class shows you how to design your app to work efficiently across different versions of Android.

To set the stage for this lesson, here is how Android's management of bitmap memory has evolved:

- On Android Android 2.2 (API level 8) and lower, when garbage

collection occurs, your app's threads get stopped. This causes a lag that can degrade performance. **Android 2.3 adds concurrent garbage collection, which means that the memory is reclaimed soon after a bitmap is no longer referenced.**
- On Android 2.3.3 (API level 10) and lower, the backing pixel data for a bitmap is stored in native memory. It is separate from the bitmap itself, which is stored in the Dalvik heap. The pixel data in native memory is not released in a predictable manner, potentially causing an application to briefly exceed its memory limits and crash. **As of Android 3.0 (API level 11), the pixel data is stored on the Dalvik heap along with the associated bitmap.**

The following sections describe how to optimize bitmap memory management for different Android versions.

Manage Memory on Android 2.3.3 and Lower

On Android 2.3.3 (API level 10) and lower, using `recycle()` is recommended. If you're displaying large amounts of bitmap data in your app, you're likely to run into `OutOfMemoryError` errors. The `recycle()` method allows an app to reclaim memory as soon as possible.

Caution: You should use `recycle()` only when you are sure that the bitmap is no longer being used. If you call `recycle()` and later attempt to draw the bitmap, you will get the error: `"Canvas: trying to use a recycled bitmap"`.

The following code snippet gives an example of calling `recycle()`. It uses reference counting (in the variables `mDisplayRefCount` and `mCacheRefCount`) to track whether a bitmap is currently being displayed or in the cache. The code recycles the bitmap when these conditions are met:

- The reference count for both `mDisplayRefCount` and `mCacheRefCount` is 0.
- The bitmap is not `null`, and it hasn't been recycled yet.

```
private int mCacheRefCount = 0;
private int mDisplayRefCount = 0;
...
// Notify the drawable that the displayed state has changed.
// Keep a count to determine when the drawable is no longer
displayed.
public void setIsDisplayed(boolean isDisplayed) {
    synchronized (this) {
        if (isDisplayed) {
            mDisplayRefCount++;
            mHasBeenDisplayed = true;
        } else {
            mDisplayRefCount--;
        }
    }
    // Check to see if recycle() can be called.
    checkState();
}

// Notify the drawable that the cache state has changed.
// Keep a count to determine when the drawable is no longer being
cached.
public void setIsCached(boolean isCached) {
    synchronized (this) {
        if (isCached) {
            mCacheRefCount++;
        } else {
            mCacheRefCount--;
        }
    }
    // Check to see if recycle() can be called.
    checkState();
}

private synchronized void checkState() {
    // If the drawable cache and display ref counts = 0, and this
drawable
    // has been displayed, then recycle.
    if (mCacheRefCount <= 0 && mDisplayRefCount <= 0 &&
mHasBeenDisplayed
            && hasValidBitmap()) {
        getBitmap().recycle();
    }
}

private synchronized boolean hasValidBitmap() {
    Bitmap bitmap = getBitmap();
    return bitmap != null && !bitmap.isRecycled();
}
```

Manage Memory on Android 3.0 and Higher

Android 3.0 (API level 11) introduces the `BitmapFactory.Options.inBitmap` field. If this option is set, decode methods that take the `Options` object will attempt to reuse an existing bitmap when loading content. This

means that the bitmap's memory is reused, resulting in improved performance, and removing both memory allocation and de-allocation. However, there are certain restrictions with how inBitmap can be used. In particular, before Android 4.4 (API level 19), only equal sized bitmaps are supported. For details, please see the inBitmap documentation.

Save a bitmap for later use

The following snippet demonstrates how an existing bitmap is stored for possible later use in the sample app. When an app is running on Android 3.0 or higher and a bitmap is evicted from the LruCache, a soft reference to the bitmap is placed in a HashSet, for possible reuse later with inBitmap:

```
Set<SoftReference<Bitmap>> mReusableBitmaps;
private LruCache<String, BitmapDrawable> mMemoryCache;

// If you're running on Honeycomb or newer, create a
// synchronized HashSet of references to reusable bitmaps.
if (Utils.hasHoneycomb()) {
    mReusableBitmaps =
            Collections.synchronizedSet(new
HashSet<SoftReference<Bitmap>>());
}

mMemoryCache = new LruCache<String,
BitmapDrawable>(mCacheParams.memCacheSize) {

    // Notify the removed entry that is no longer being cached.
    @Override
    protected void entryRemoved(boolean evicted, String key,
            BitmapDrawable oldValue, BitmapDrawable newValue) {
        if (RecyclingBitmapDrawable.class.isInstance(oldValue)) {
            // The removed entry is a recycling drawable, so notify
it
            // that it has been removed from the memory cache.
            ((RecyclingBitmapDrawable) oldValue).setIsCached(false);
        } else {
            // The removed entry is a standard BitmapDrawable.
            if (Utils.hasHoneycomb()) {
                // We're running on Honeycomb or later, so add the
bitmap
                // to a SoftReference set for possible use with
inBitmap later.
                mReusableBitmaps.add
                        (new
SoftReference<Bitmap>(oldValue.getBitmap()));
            }
        }
    }
....
}
```

Use an existing bitmap

In the running app, decoder methods check to see if there is an existing bitmap they can use. For example:

```
public static Bitmap decodeSampledBitmapFromFile(String filename,
        int reqWidth, int reqHeight, ImageCache cache) {

    final BitmapFactory.Options options = new
BitmapFactory.Options();
    ...
    BitmapFactory.decodeFile(filename, options);
    ...

    // If we're running on Honeycomb or newer, try to use inBitmap.
    if (Utils.hasHoneycomb()) {
        addInBitmapOptions(options, cache);
    }
    ...
    return BitmapFactory.decodeFile(filename, options);
}
```

The next snippet shows the `addInBitmapOptions()` method that is called in the above snippet. It looks for an existing bitmap to set as the value for `inBitmap`. Note that this method only sets a value for `inBitmap` if it finds a suitable match (your code should never assume that a match will be found):

```
private static void addInBitmapOptions(BitmapFactory.Options options,
        ImageCache cache) {
    // inBitmap only works with mutable bitmaps, so force the
decoder to
    // return mutable bitmaps.
    options.inMutable = true;

    if (cache != null) {
        // Try to find a bitmap to use for inBitmap.
        Bitmap inBitmap = cache.getBitmapFromReusableSet(options);

        if (inBitmap != null) {
            // If a suitable bitmap has been found, set it as the
value of
            // inBitmap.
            options.inBitmap = inBitmap;
        }
    }
}

// This method iterates through the reusable bitmaps, looking for
one
// to use for inBitmap:
protected Bitmap getBitmapFromReusableSet(BitmapFactory.Options
options) {
```

```
        Bitmap bitmap = null;

    if (mReusableBitmaps != null && !mReusableBitmaps.isEmpty()) {
        synchronized (mReusableBitmaps) {
            final Iterator<SoftReference<Bitmap>> iterator
                    = mReusableBitmaps.iterator();
            Bitmap item;

            while (iterator.hasNext()) {
                item = iterator.next().get();

                if (null != item && item.isMutable()) {
                    // Check to see it the item can be used for
inBitmap.
                    if (canUseForInBitmap(item, options)) {
                        bitmap = item;

                        // Remove from reusable set so it can't be
used again.
                        iterator.remove();
                        break;
                    }
                } else {
                    // Remove from the set if the reference has been
cleared.
                    iterator.remove();
                }
            }
        }
    }
    return bitmap;
}
```

Finally, this method determines whether a candidate bitmap satisfies the size criteria to be used for `inBitmap`:

```
static boolean canUseForInBitmap(
        Bitmap candidate, BitmapFactory.Options targetOptions) {

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.KITKAT) {
        // From Android 4.4 (KitKat) onward we can re-use if the
byte size of
        // the new bitmap is smaller than the reusable bitmap
candidate
        // allocation byte count.
        int width = targetOptions.outWidth /
targetOptions.inSampleSize;
        int height = targetOptions.outHeight /
targetOptions.inSampleSize;
        int byteCount = width * height *
getBytesPerPixel(candidate.getConfig());
        return byteCount <= candidate.getAllocationByteCount();
    }

    // On earlier versions, the dimensions must match exactly and
the inSampleSize must be 1
    return candidate.getWidth() == targetOptions.outWidth
            && candidate.getHeight() == targetOptions.outHeight
            && targetOptions.inSampleSize == 1;
```

```
}

/**
 * A helper function to return the byte usage per pixel of a bitmap
based on its configuration.
 */
static int getBytesPerPixel(Config config) {
    if (config == Config.ARGB_8888) {
        return 4;
    } else if (config == Config.RGB_565) {
        return 2;
    } else if (config == Config.ARGB_4444) {
        return 2;
    } else if (config == Config.ALPHA_8) {
        return 1;
    }
    return 1;
}
```

Next Get started Next class

Displaying Bitmaps in Your UI

Previous Next Get started

This lesson teaches you to

1. Load Bitmaps into a ViewPager Implementation
2. Load Bitmaps into a GridView Implementation

You should also read

- Android Design: Swipe Views
- Android Design: Grid Lists

Try it out

Download the sample

DisplayingBitmaps.zip

This lesson brings together everything from previous lessons, showing you how to load multiple bitmaps into `ViewPager` and `GridView` components using a background thread and bitmap cache, while dealing with concurrency and configuration changes.

Load Bitmaps into a ViewPager Implementation

The swipe view pattern is an excellent way to navigate the detail view of an image gallery. You can implement this pattern using a `ViewPager` component backed by a `PagerAdapter`. However, a more suitable backing adapter is the subclass `FragmentStatePagerAdapter` which automatically

destroys and saves state of the Fragments in the ViewPager as they disappear off-screen, keeping memory usage down.

Note: If you have a smaller number of images and are confident they all fit within the application memory limit, then using a regular PagerAdapter or FragmentPagerAdapter might be more appropriate.

Here's an implementation of a ViewPager with ImageView children. The main activity holds the ViewPager and the adapter:

```
public class ImageDetailActivity extends FragmentActivity {
    public static final String EXTRA_IMAGE = "extra_image";

    private ImagePagerAdapter mAdapter;
    private ViewPager mPager;

    // A static dataset to back the ViewPager adapter
    public final static Integer[] imageResIds = new Integer[] {
            R.drawable.sample_image_1, R.drawable.sample_image_2,
R.drawable.sample_image_3,
            R.drawable.sample_image_4, R.drawable.sample_image_5,
R.drawable.sample_image_6,
            R.drawable.sample_image_7, R.drawable.sample_image_8,
R.drawable.sample_image_9};

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.image_detail_pager); // Contains just
a ViewPager

        mAdapter = new ImagePagerAdapter(getSupportFragmentManager(),
imageResIds.length);
        mPager = (ViewPager) findViewById(R.id.pager);
        mPager.setAdapter(mAdapter);
    }

    public static class ImagePagerAdapter extends
FragmentStatePagerAdapter {
        private final int mSize;

        public ImagePagerAdapter(FragmentManager fm, int size) {
            super(fm);
            mSize = size;
        }

        @Override
        public int getCount() {
            return mSize;
        }

        @Override
        public Fragment getItem(int position) {
            return ImageDetailFragment.newInstance(position);
        }
```

```
    }
}
```

Here is an implementation of the details `Fragment` which holds the `ImageView` children. This might seem like a perfectly reasonable approach, but can you see the drawbacks of this implementation? How could it be improved?

```
public class ImageDetailFragment extends Fragment {
    private static final String IMAGE_DATA_EXTRA = "resId";
    private int mImageNum;
    private ImageView mImageView;

    static ImageDetailFragment newInstance(int imageNum) {
        final ImageDetailFragment f = new ImageDetailFragment();
        final Bundle args = new Bundle();
        args.putInt(IMAGE_DATA_EXTRA, imageNum);
        f.setArguments(args);
        return f;
    }

    // Empty constructor, required as per Fragment docs
    public ImageDetailFragment() {}

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mImageNum = getArguments() != null ?
getArguments().getInt(IMAGE_DATA_EXTRA) : -1;
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup
container,
            Bundle savedInstanceState) {
        // image_detail_fragment.xml contains just an ImageView
        final View v =
inflater.inflate(R.layout.image_detail_fragment, container, false);
        mImageView = (ImageView) v.findViewById(R.id.imageView);
        return v;
    }

    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        final int resId = ImageDetailActivity.imageResIds[mImageNum];
        mImageView.setImageResource(resId); // Load image into
ImageView
    }
}
```

Hopefully you noticed the issue: the images are being read from resources on the UI thread, which can lead to an application hanging and being force closed. Using an `AsyncTask` as described in the Processing Bitmaps Off the UI Thread lesson, it's straightforward to move image

loading and processing to a background thread:

```
public class ImageDetailActivity extends FragmentActivity {
    ...

    public void loadBitmap(int resId, ImageView imageView) {
        mImageView.setImageResource(R.drawable.image_placeholder);
        BitmapWorkerTask task = new BitmapWorkerTask(mImageView);
        task.execute(resId);
    }

    ... // include BitmapWorkerTask class
}

public class ImageDetailFragment extends Fragment {
    ...

    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        if (ImageDetailActivity.class.isInstance(getActivity())) {
            final int resId =
ImageDetailActivity.imageResIds[mImageNum];
            // Call out to ImageDetailActivity to load the bitmap in
a background thread
            ((ImageDetailActivity) getActivity()).loadBitmap(resId,
mImageView);
        }
    }
}
```

Any additional processing (such as resizing or fetching images from the network) can take place in the `BitmapWorkerTask` without affecting responsiveness of the main UI. If the background thread is doing more than just loading an image directly from disk, it can also be beneficial to add a memory and/or disk cache as described in the lesson Caching Bitmaps. Here's the additional modifications for a memory cache:

```
public class ImageDetailActivity extends FragmentActivity {
    ...
    private LruCache<String, Bitmap> mMemoryCache;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        ...
        // initialize LruCache as per Use a Memory Cache section
    }

    public void loadBitmap(int resId, ImageView imageView) {
        final String imageKey = String.valueOf(resId);

        final Bitmap bitmap = mMemoryCache.get(imageKey);
        if (bitmap != null) {
            mImageView.setImageBitmap(bitmap);
        } else {
```

```
mImageView.setImageResource(R.drawable.image_placeholder);
            BitmapWorkerTask task = new BitmapWorkerTask(mImageView);
            task.execute(resId);
        }
    }

    ... // include updated BitmapWorkerTask from Use a Memory Cache
section
}
```

Putting all these pieces together gives you a responsive `ViewPager` implementation with minimal image loading latency and the ability to do as much or as little background processing on your images as needed.

Load Bitmaps into a GridView Implementation

The grid list building block is useful for showing image data sets and can be implemented using a `GridView` component in which many images can be on-screen at any one time and many more need to be ready to appear if the user scrolls up or down. When implementing this type of control, you must ensure the UI remains fluid, memory usage remains under control and concurrency is handled correctly (due to the way `GridView` recycles its children views).

To start with, here is a standard `GridView` implementation with `ImageView` children placed inside a `Fragment`. Again, this might seem like a perfectly reasonable approach, but what would make it better?

```
public class ImageGridFragment extends Fragment implements
AdapterView.OnItemClickListener {
    private ImageAdapter mAdapter;

    // A static dataset to back the GridView adapter
    public final static Integer[] imageResIds = new Integer[] {
            R.drawable.sample_image_1, R.drawable.sample_image_2,
R.drawable.sample_image_3,
            R.drawable.sample_image_4, R.drawable.sample_image_5,
R.drawable.sample_image_6,
            R.drawable.sample_image_7, R.drawable.sample_image_8,
R.drawable.sample_image_9};

    // Empty constructor as per Fragment docs
    public ImageGridFragment() {}

    @Override
    public void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);
        mAdapter = new ImageAdapter(getActivity());
    }

    @Override
    public View onCreateView(
            LayoutInflater inflater, ViewGroup container, Bundle
savedInstanceState) {
        final View v = inflater.inflate(R.layout.image_grid_fragment,
container, false);
        final GridView mGridView = (GridView)
v.findViewById(R.id.gridView);
        mGridView.setAdapter(mAdapter);
        mGridView.setOnItemClickListener(this);
        return v;
    }

    @Override
    public void onItemClick(AdapterView<?> parent, View v, int
position, long id) {
        final Intent i = new Intent(getActivity(),
ImageDetailActivity.class);
        i.putExtra(ImageDetailActivity.EXTRA_IMAGE, position);
        startActivity(i);
    }

    private class ImageAdapter extends BaseAdapter {
        private final Context mContext;

        public ImageAdapter(Context context) {
            super();
            mContext = context;
        }

        @Override
        public int getCount() {
            return imageResIds.length;
        }

        @Override
        public Object getItem(int position) {
            return imageResIds[position];
        }

        @Override
        public long getItemId(int position) {
            return position;
        }

        @Override
        public View getView(int position, View convertView,
ViewGroup container) {
            ImageView imageView;
            if (convertView == null) { // if it's not recycled,
initialize some attributes
                imageView = new ImageView(mContext);

imageView.setScaleType(ImageView.ScaleType.CENTER_CROP);
                imageView.setLayoutParams(new GridView.LayoutParams(
                        LayoutParams.MATCH_PARENT,
LayoutParams.MATCH_PARENT));
```

```
            } else {
                imageView = (ImageView) convertView;
            }
            imageView.setImageResource(imageResIds[position]); //
Load image into ImageView
            return imageView;
        }
    }
}
```

Once again, the problem with this implementation is that the image is being set in the UI thread. While this may work for small, simple images (due to system resource loading and caching), if any additional processing needs to be done, your UI grinds to a halt.

The same asynchronous processing and caching methods from the previous section can be implemented here. However, you also need to wary of concurrency issues as the `GridView` recycles its children views. To handle this, use the techniques discussed in the Processing Bitmaps Off the UI Thread lesson. Here is the updated solution:

```
public class ImageGridFragment extends Fragment implements
AdapterView.OnItemClickListener {
    ...

    private class ImageAdapter extends BaseAdapter {
        ...

        @Override
        public View getView(int position, View convertView,
ViewGroup container) {
            ...
            loadBitmap(imageResIds[position], imageView)
            return imageView;
        }
    }

    public void loadBitmap(int resId, ImageView imageView) {
        if (cancelPotentialWork(resId, imageView)) {
            final BitmapWorkerTask task = new
BitmapWorkerTask(imageView);
            final AsyncDrawable asyncDrawable =
                    new AsyncDrawable(getResources(),
mPlaceHolderBitmap, task);
            imageView.setImageDrawable(asyncDrawable);
            task.execute(resId);
        }
    }

    static class AsyncDrawable extends BitmapDrawable {
        private final WeakReference<BitmapWorkerTask>
bitmapWorkerTaskReference;

        public AsyncDrawable(Resources res, Bitmap bitmap,
```

```
                BitmapWorkerTask bitmapWorkerTask) {
            super(res, bitmap);
            bitmapWorkerTaskReference =
                new
WeakReference<BitmapWorkerTask>(bitmapWorkerTask);
        }

        public BitmapWorkerTask getBitmapWorkerTask() {
            return bitmapWorkerTaskReference.get();
        }
    }

    public static boolean cancelPotentialWork(int data, ImageView
imageView) {
        final BitmapWorkerTask bitmapWorkerTask =
getBitmapWorkerTask(imageView);

        if (bitmapWorkerTask != null) {
            final int bitmapData = bitmapWorkerTask.data;
            if (bitmapData != data) {
                // Cancel previous task
                bitmapWorkerTask.cancel(true);
            } else {
                // The same work is already in progress
                return false;
            }
        }
        // No task associated with the ImageView, or an existing
task was cancelled
        return true;
    }

    private static BitmapWorkerTask getBitmapWorkerTask(ImageView
imageView) {
       if (imageView != null) {
           final Drawable drawable = imageView.getDrawable();
           if (drawable instanceof AsyncDrawable) {
               final AsyncDrawable asyncDrawable = (AsyncDrawable)
drawable;
               return asyncDrawable.getBitmapWorkerTask();
           }
        }
        return null;
    }

    ... // include updated BitmapWorkerTask class
```

Note: The same code can easily be adapted to work with `ListView` as well.

This implementation allows for flexibility in how the images are processed and loaded without impeding the smoothness of the UI. In the background task you can load images from the network or resize large digital camera photos and the images appear as the tasks finish processing.

For a full example of this and other concepts discussed in this lesson, please see the included sample application.

Next Get started Next class

Displaying Graphics with OpenGL ES

Previous Next Get started

Dependencies and prerequisites

- Android 2.2 (API Level 8) or higher
- Experience building an Android app

You should also read

- OpenGL

Try it out

Download the sample

OpenGLES.zip

The Android framework provides plenty of standard tools for creating attractive, functional graphical user interfaces. However, if you want more control of what your application draws on screen, or are venturing into three dimensional graphics, you need to use a different tool. The OpenGL ES APIs provided by the Android framework offers a set of tools for displaying high-end, animated graphics that are limited only by your imagination and can also benefit from the acceleration of graphics processing units (GPUs) provided on many Android devices.

This class walks you through the basics of developing applications that use OpenGL, including setup, drawing objects, moving drawn elements and responding to touch input.

The example code in this class uses the OpenGL ES 2.0 APIs, which is the recommended API version to use with current Android devices. For more information about versions of OpenGL ES, see the OpenGL developer guide.

Note: Be careful not to mix OpenGL ES 1.x API calls with OpenGL ES 2.0 methods! The two APIs are not interchangeable and trying to use them together only results in frustration and sadness.

Lessons

Building an OpenGL ES Environment
: Learn how to set up an Android application to be able to draw OpenGL graphics.

Defining Shapes
: Learn how to define shapes and why you need to know about faces and winding.

Drawing Shapes
: Learn how to draw OpenGL shapes in your application.

Applying Projection and Camera Views
: Learn how to use projection and camera views to get a new perspective on your drawn objects.

Adding Motion
: Learn how to do basic movement and animation of drawn objects with OpenGL.

Responding to Touch Events
: Learn how to do basic interaction with OpenGL graphics.

Next Get started Next class

Building an OpenGL ES Environment

Previous Next Get started

This lesson teaches you to

1. Declare OpenGL ES Use in the Manifest
2. Create an Activity for OpenGL ES Graphics
3. Build a GLSurfaceView Object
4. Build a Renderer Class

You should also read

- OpenGL

Try it out

Download the sample

OpenGLES.zip

In order to draw graphics with OpenGL ES in your Android application, you must create a view container for them. One of the more straight-forward ways to do this is to implement both a `GLSurfaceView` and a `GLSurfaceView.Renderer`. A `GLSurfaceView` is a view container for graphics drawn with OpenGL and `GLSurfaceView.Renderer` controls what is drawn within that view. For more information about these classes, see the OpenGL ES developer guide.

`GLSurfaceView` is just one way to incorporate OpenGL ES graphics into your application. For a full-screen or near-full screen graphics view, it is a

reasonable choice. Developers who want to incorporate OpenGL ES graphics in a small portion of their layouts should take a look at `TextureView`. For real, do-it-yourself developers, it is also possible to build up an OpenGL ES view using `SurfaceView`, but this requires writing quite a bit of additional code.

This lesson explains how to complete a minimal implementation of `GLSurfaceView` and `GLSurfaceView.Renderer` in a simple application activity.

Declare OpenGL ES Use in the Manifest

In order for your application to use the OpenGL ES 2.0 API, you must add the following declaration to your manifest:

```
<uses-feature android:glEsVersion="0x00020000"
android:required="true" />
```

If your application uses texture compression, you must also declare which compression formats your app supports, so that it is only installed on compatible devices.

```
<supports-gl-texture
android:name="GL_OES_compressed_ETC1_RGB8_texture" />
<supports-gl-texture
android:name="GL_OES_compressed_paletted_texture" />
```

For more information about texture compression formats, see the OpenGL developer guide.

Create an Activity for OpenGL ES Graphics

Android applications that use OpenGL ES have activities just like any other application that has a user interface. The main difference from other applications is what you put in the layout for your activity. While in many applications you might use `TextView`, `Button` and `ListView`, in an app that uses OpenGL ES, you can also add a `GLSurfaceView`.

The following code example shows a minimal implementation of an activity that uses a `GLSurfaceView` as its primary view:

```
public class OpenGLES20Activity extends Activity {

    private GLSurfaceView mGLView;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Create a GLSurfaceView instance and set it
        // as the ContentView for this Activity.
        mGLView = new MyGLSurfaceView(this);
        setContentView(mGLView);
    }
}
```

Note: OpenGL ES 2.0 requires Android 2.2 (API Level 8) or higher, so make sure your Android project targets that API or higher.

Build a GLSurfaceView Object

A `GLSurfaceView` is a specialized view where you can draw OpenGL ES graphics. It does not do much by itself. The actual drawing of objects is controlled in the `GLSurfaceView.Renderer` that you set on this view. In fact, the code for this object is so thin, you may be tempted to skip extending it and just create an unmodified `GLSurfaceView` instance, but don't do that. You need to extend this class in order to capture touch events, which is covered in the Responding to Touch Events lesson.

The essential code for a `GLSurfaceView` is minimal, so for a quick implementation, it is common to just create an inner class in the activity that uses it:

```
class MyGLSurfaceView extends GLSurfaceView {

    private final MyGLRenderer mRenderer;

    public MyGLSurfaceView(Context context){
        super(context);

        // Create an OpenGL ES 2.0 context
        setEGLContextClientVersion(2);

        mRenderer = new MyGLRenderer();

        // Set the Renderer for drawing on the GLSurfaceView
        setRenderer(mRenderer);
    }
}
```

One other optional addition to your GLSurfaceView implementation is to set the render mode to only draw the view when there is a change to your drawing data using the GLSurfaceView.RENDERMODE_WHEN_DIRTY setting:

```
// Render the view only when there is a change in the drawing data
setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
```

This setting prevents the GLSurfaceView frame from being redrawn until you call requestRender(), which is more efficient for this sample app.

Build a Renderer Class

The implementation of the GLSurfaceView.Renderer class, or renderer, within an application that uses OpenGL ES is where things start to get interesting. This class controls what gets drawn on the GLSurfaceView with which it is associated. There are three methods in a renderer that are called by the Android system in order to figure out what and how to draw on a GLSurfaceView:

- onSurfaceCreated() - Called once to set up the view's OpenGL ES environment.
- onDrawFrame() - Called for each redraw of the view.
- onSurfaceChanged() - Called if the geometry of the view changes, for example when the device's screen orientation changes.

Here is a very basic implementation of an OpenGL ES renderer, that does nothing more than draw a black background in the GLSurfaceView:

```
public class MyGLRenderer implements GLSurfaceView.Renderer {

    public void onSurfaceCreated(GL10 unused, EGLConfig config) {
        // Set the background frame color
        GLES20.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    }

    public void onDrawFrame(GL10 unused) {
        // Redraw background color
        GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT);
    }

    public void onSurfaceChanged(GL10 unused, int width, int height)
{
        GLES20.glViewport(0, 0, width, height);
    }
```

```
}
```

That's all there is to it! The code examples above create a simple Android application that displays a black screen using OpenGL. While this code does not do anything very interesting, by creating these classes, you have laid the foundation you need to start drawing graphic elements with OpenGL.

Note: You may wonder why these methods have a `GL10` parameter, when you are using the OpengGL ES 2.0 APIs. These method signatures are simply reused for the 2.0 APIs to keep the Android framework code simpler.

If you are familiar with the OpenGL ES APIs, you should now be able to set up a OpenGL ES environment in your app and start drawing graphics. However, if you need a bit more help getting started with OpenGL, head on to the next lessons for a few more hints.

Next Get started Next class

Defining Shapes

Previous Next Get started

This lesson teaches you to

1. Define a Triangle
2. Define a Square

You should also read

- OpenGL

Download the sample

OpenGLES.zip

Being able to define shapes to be drawn in the context of an OpenGL ES view is the first step in creating your high-end graphics masterpiece. Drawing with OpenGL ES can be a little tricky without knowing a few basic things about how OpenGL ES expects you to define graphic objects.

This lesson explains the OpenGL ES coordinate system relative to an Android device screen, the basics of defining a shape, shape faces, as well as defining a triangle and a square.

Define a Triangle

OpenGL ES allows you to define drawn objects using coordinates in three-dimensional space. So, before you can draw a triangle, you must define its coordinates. In OpenGL, the typical way to do this is to define a vertex array of floating point numbers for the coordinates. For maximum efficiency, you write these coordinates into a `ByteBuffer`, that is passed

into the OpenGL ES graphics pipeline for processing.

```
public class Triangle {

    private FloatBuffer vertexBuffer;

    // number of coordinates per vertex in this array
    static final int COORDS_PER_VERTEX = 3;
    static float triangleCoords[] = {   // in counterclockwise
order:
             0.0f,  0.622008459f, 0.0f, // top
            -0.5f, -0.311004243f, 0.0f, // bottom left
             0.5f, -0.311004243f, 0.0f  // bottom right
    };

    // Set color with red, green, blue and alpha (opacity) values
    float color[] = { 0.63671875f, 0.76953125f, 0.22265625f, 1.0f };

    public Triangle() {
        // initialize vertex byte buffer for shape coordinates
        ByteBuffer bb = ByteBuffer.allocateDirect(
                // (number of coordinate values * 4 bytes per float)
                triangleCoords.length * 4);
        // use the device hardware's native byte order
        bb.order(ByteOrder.nativeOrder());

        // create a floating point buffer from the ByteBuffer
        vertexBuffer = bb.asFloatBuffer();
        // add the coordinates to the FloatBuffer
        vertexBuffer.put(triangleCoords);
        // set the buffer to read the first coordinate
        vertexBuffer.position(0);
    }
}
```

By default, OpenGL ES assumes a coordinate system where [0,0,0] (X,Y,Z) specifies the center of the `GLSurfaceView` frame, [1,1,0] is the top right corner of the frame and [-1,-1,0] is bottom left corner of the frame. For an illustration of this coordinate system, see the OpenGL ES developer guide.

Note that the coordinates of this shape are defined in a counterclockwise order. The drawing order is important because it defines which side is the front face of the shape, which you typically want to have drawn, and the back face, which you can choose to not draw using the OpenGL ES cull face feature. For more information about faces and culling, see the OpenGL ES developer guide.

Define a Square

Defining triangles is pretty easy in OpenGL, but what if you want to get a just a little more complex? Say, a square? There are a number of ways to do this, but a typical path to drawing such a shape in OpenGL ES is to use two triangles drawn together:

Figure 1. Drawing a square using two triangles.

Again, you should define the vertices in a counterclockwise order for both triangles that represent this shape, and put the values in a `ByteBuffer`. In order to avoid defining the two coordinates shared by each triangle twice, use a drawing list to tell the OpenGL ES graphics pipeline how to draw these vertices. Here's the code for this shape:

```
public class Square {

    private FloatBuffer vertexBuffer;
    private ShortBuffer drawListBuffer;

    // number of coordinates per vertex in this array
    static final int COORDS_PER_VERTEX = 3;
    static float squareCoords[] = {
            -0.5f,  0.5f, 0.0f,   // top left
            -0.5f, -0.5f, 0.0f,   // bottom left
             0.5f, -0.5f, 0.0f,   // bottom right
             0.5f,  0.5f, 0.0f }; // top right

    private short drawOrder[] = { 0, 1, 2, 0, 2, 3 }; // order to
draw vertices

    public Square() {
        // initialize vertex byte buffer for shape coordinates
        ByteBuffer bb = ByteBuffer.allocateDirect(
        // (# of coordinate values * 4 bytes per float)
                squareCoords.length * 4);
```

```
        bb.order(ByteOrder.nativeOrder());
        vertexBuffer = bb.asFloatBuffer();
        vertexBuffer.put(squareCoords);
        vertexBuffer.position(0);

        // initialize byte buffer for the draw list
        ByteBuffer dlb = ByteBuffer.allocateDirect(
        // (# of coordinate values * 2 bytes per short)
                drawOrder.length * 2);
        dlb.order(ByteOrder.nativeOrder());
        drawListBuffer = dlb.asShortBuffer();
        drawListBuffer.put(drawOrder);
        drawListBuffer.position(0);
    }
}
```

This example gives you a peek at what it takes to create more complex shapes with OpenGL. In general, you use collections of triangles to draw objects. In the next lesson, you learn how to draw these shapes on screen.

Next Get started Next class

Drawing Shapes

Previous Next Get started

This lesson teaches you to

1. Initialize Shapes
2. Draw a Shape

You should also read

- OpenGL

Download the sample

OpenGLES.zip

After you define shapes to be drawn with OpenGL, you probably want to draw them. Drawing shapes with the OpenGL ES 2.0 takes a bit more code than you might imagine, because the API provides a great deal of control over the graphics rendering pipeline.

This lesson explains how to draw the shapes you defined in the previous lesson using the OpenGL ES 2.0 API.

Initialize Shapes

Before you do any drawing, you must initialize and load the shapes you plan to draw. Unless the structure (the original coordinates) of the shapes you use in your program change during the course of execution, you should initialize them in the `onSurfaceCreated()` method of your renderer for memory and processing efficiency.

```
public class MyGLRenderer implements GLSurfaceView.Renderer {
```

```
    ...
    private Triangle mTriangle;
    private Square   mSquare;

    public void onSurfaceCreated(GL10 unused, EGLConfig config) {
        ...

        // initialize a triangle
        mTriangle = new Triangle();
        // initialize a square
        mSquare = new Square();
    }
    ...
}
```

Draw a Shape

Drawing a defined shape using OpenGL ES 2.0 requires a significant amount of code, because you must provide a lot of details to the graphics rendering pipeline. Specifically, you must define the following:

- *Vertex Shader* - OpenGL ES graphics code for rendering the vertices of a shape.
- *Fragment Shader* - OpenGL ES code for rendering the face of a shape with colors or textures.
- *Program* - An OpenGL ES object that contains the shaders you want to use for drawing one or more shapes.

You need at least one vertex shader to draw a shape and one fragment shader to color that shape. These shaders must be complied and then added to an OpenGL ES program, which is then used to draw the shape. Here is an example of how to define basic shaders you can use to draw a shape in the `Triangle` class:

```
public class Triangle {

    private final String vertexShaderCode =
        "attribute vec4 vPosition;" +
        "void main() {" +
        "  gl_Position = vPosition;" +
        "}";

    private final String fragmentShaderCode =
        "precision mediump float;" +
        "uniform vec4 vColor;" +
        "void main() {" +
        "  gl_FragColor = vColor;" +
        "}";
```

```
    ...
}
```

Shaders contain OpenGL Shading Language (GLSL) code that must be compiled prior to using it in the OpenGL ES environment. To compile this code, create a utility method in your renderer class:

```
public static int loadShader(int type, String shaderCode){

    // create a vertex shader type (GLES20.GL_VERTEX_SHADER)
    // or a fragment shader type (GLES20.GL_FRAGMENT_SHADER)
    int shader = GLES20.glCreateShader(type);

    // add the source code to the shader and compile it
    GLES20.glShaderSource(shader, shaderCode);
    GLES20.glCompileShader(shader);

    return shader;
}
```

In order to draw your shape, you must compile the shader code, add them to a OpenGL ES program object and then link the program. Do this in your drawn object's constructor, so it is only done once.

Note: Compiling OpenGL ES shaders and linking programs is expensive in terms of CPU cycles and processing time, so you should avoid doing this more than once. If you do not know the content of your shaders at runtime, you should build your code such that they only get created once and then cached for later use.

```
public class Triangle() {
    ...

    private final int mProgram;

    public Triangle() {
        ...

        int vertexShader =
MyGLRenderer.loadShader(GLES20.GL_VERTEX_SHADER,
                                        vertexShaderCode);
        int fragmentShader =
MyGLRenderer.loadShader(GLES20.GL_FRAGMENT_SHADER,
                                        fragmentShaderCode);

        // create empty OpenGL ES Program
        mProgram = GLES20.glCreateProgram();

        // add the vertex shader to program
        GLES20.glAttachShader(mProgram, vertexShader);

        // add the fragment shader to program
        GLES20.glAttachShader(mProgram, fragmentShader);
```

```
        // creates OpenGL ES program executables
        GLES20.glLinkProgram(mProgram);
    }
}
```

At this point, you are ready to add the actual calls that draw your shape. Drawing shapes with OpenGL ES requires that you specify several parameters to tell the rendering pipeline what you want to draw and how to draw it. Since drawing options can vary by shape, it's a good idea to have your shape classes contain their own drawing logic.

Create a `draw()` method for drawing the shape. This code sets the position and color values to the shape's vertex shader and fragment shader, and then executes the drawing function.

```
private int mPositionHandle;
private int mColorHandle;

private final int vertexCount = triangleCoords.length /
COORDS_PER_VERTEX;
private final int vertexStride = COORDS_PER_VERTEX * 4; // 4 bytes
per vertex

public void draw() {
    // Add program to OpenGL ES environment
    GLES20.glUseProgram(mProgram);

    // get handle to vertex shader's vPosition member
    mPositionHandle = GLES20.glGetAttribLocation(mProgram,
"vPosition");

    // Enable a handle to the triangle vertices
    GLES20.glEnableVertexAttribArray(mPositionHandle);

    // Prepare the triangle coordinate data
    GLES20.glVertexAttribPointer(mPositionHandle, COORDS_PER_VERTEX,
                                 GLES20.GL_FLOAT, false,
                                 vertexStride, vertexBuffer);

    // get handle to fragment shader's vColor member
    mColorHandle = GLES20.glGetUniformLocation(mProgram, "vColor");

    // Set color for drawing the triangle
    GLES20.glUniform4fv(mColorHandle, 1, color, 0);

    // Draw the triangle
    GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, vertexCount);

    // Disable vertex array
    GLES20.glDisableVertexAttribArray(mPositionHandle);
}
```

Once you have all this code in place, drawing this object just requires a

call to the `draw()` method from within your renderer's `onDrawFrame()` method:

```
public void onDrawFrame(GL10 unused) {
    ...

    mTriangle.draw();
}
```

When you run the application, it should look something like this:

Figure 1. Triangle drawn without a projection or camera view.

There are a few problems with this code example. First of all, it is not going to impress your friends. Secondly, the triangle is a bit squashed and changes shape when you change the screen orientation of the device. The reason the shape is skewed is due to the fact that the object's vertices have not been corrected for the proportions of the screen area where the `GLSurfaceView` is displayed. You can fix that problem using a projection and camera view in the next lesson.

Lastly, the triangle is stationary, which is a bit boring. In the Adding Motion lesson, you make this shape rotate and make more interesting use of the OpenGL ES graphics pipeline.

Next Get started Next class

Applying Projection and Camera Views

Previous Next Get started

This lesson teaches you to

1. Define a Projection
2. Define a Camera View
3. Apply Projection and Camera Transformations

You should also read

- OpenGL

Download the sample

OpenGLES.zip

In the OpenGL ES environment, projection and camera views allow you to display drawn objects in a way that more closely resembles how you see physical objects with your eyes. This simulation of physical viewing is done with mathematical transformations of drawn object coordinates:

- *Projection* - This transformation adjusts the coordinates of drawn objects based on the width and height of the `GLSurfaceView` where they are displayed. Without this calculation, objects drawn by OpenGL ES are skewed by the unequal proportions of the view window. A projection transformation typically only has to be calculated when the proportions of the OpenGL view are established or changed in the `onSurfaceChanged()` method of your renderer. For more information about OpenGL ES projections and coordinate mapping, see Mapping Coordinates for Drawn Objects.

- *Camera View* - This transformation adjusts the coordinates of drawn objects based on a virtual camera position. It's important to note that OpenGL ES does not define an actual camera object, but instead provides utility methods that simulate a camera by transforming the display of drawn objects. A camera view transformation might be calculated only once when you establish your `GLSurfaceView`, or might change dynamically based on user actions or your application's function.

This lesson describes how to create a projection and camera view and apply it to shapes drawn in your `GLSurfaceView`.

Define a Projection

The data for a projection transformation is calculated in the `onSurfaceChanged()` method of your `GLSurfaceView.Renderer` class. The following example code takes the height and width of the `GLSurfaceView` and uses it to populate a projection transformation `Matrix` using the `Matrix.frustumM()` method:

```
// mMVPMatrix is an abbreviation for "Model View Projection Matrix"
private final float[] mMVPMatrix = new float[16];
private final float[] mProjectionMatrix = new float[16];
private final float[] mViewMatrix = new float[16];

@Override
public void onSurfaceChanged(GL10 unused, int width, int height) {
    GLES20.glViewport(0, 0, width, height);

    float ratio = (float) width / height;

    // this projection matrix is applied to object coordinates
    // in the onDrawFrame() method
    Matrix.frustumM(mProjectionMatrix, 0, -ratio, ratio, -1, 1, 3,
7);
}
```

This code populates a projection matrix, `mProjectionMatrix` which you can then combine with a camera view transformation in the `onDrawFrame()` method, which is shown in the next section.

Note: Just applying a projection transformation to your drawing objects typically results in a very empty display. In general, you must also apply a camera view transformation in order for anything to show up on screen.

Define a Camera View

Complete the process of transforming your drawn objects by adding a camera view transformation as part of the drawing process in your renderer. In the following example code, the camera view transformation is calculated using the `Matrix.setLookAtM()` method and then combined with the previously calculated projection matrix. The combined transformation matrices are then passed to the drawn shape.

```
@Override
public void onDrawFrame(GL10 unused) {
    ...
    // Set the camera position (View matrix)
    Matrix.setLookAtM(mViewMatrix, 0, 0, 0, -3, 0f, 0f, 0f, 0f, 1.0f,
0.0f);

    // Calculate the projection and view transformation
    Matrix.multiplyMM(mMVPMatrix, 0, mProjectionMatrix, 0,
mViewMatrix, 0);

    // Draw shape
    mTriangle.draw(mMVPMatrix);
}
```

Apply Projection and Camera Transformations

In order to use the combined projection and camera view transformation matrix shown in the previews sections, first add a matrix variable to the *vertex shader* previously defined in the `Triangle` class:

```
public class Triangle {

    private final String vertexShaderCode =
        // This matrix member variable provides a hook to manipulate
        // the coordinates of the objects that use this vertex shader
        "uniform mat4 uMVPMatrix;" +
        "attribute vec4 vPosition;" +
        "void main() {" +
        // the matrix must be included as a modifier of gl_Position
        // Note that the uMVPMatrix factor *must be first* in order
        // for the matrix multiplication product to be correct.
        "  gl_Position = uMVPMatrix * vPosition;" +
        "}";

    // Use to access and set the view transformation
    private int mMVPMatrixHandle;
```

```
    ...
}
```

Next, modify the `draw()` method of your graphic objects to accept the combined transformation matrix and apply it to the shape:

```
public void draw(float[] mvpMatrix) { // pass in the calculated
transformation matrix
    ...

    // get handle to shape's transformation matrix
    mMVPMatrixHandle = GLES20.glGetUniformLocation(mProgram,
"uMVPMatrix");

    // Pass the projection and view transformation to the shader
    GLES20.glUniformMatrix4fv(mMVPMatrixHandle, 1, false, mvpMatrix,
0);

    // Draw the triangle
    GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, vertexCount);

    // Disable vertex array
    GLES20.glDisableVertexAttribArray(mPositionHandle);
}
```

Once you have correctly calculated and applied the projection and camera view transformations, your graphic objects are drawn in correct proportions and should look like this:

Figure 1. Triangle drawn with a projection and camera view applied.

Now that you have an application that displays your shapes in correct proportions, it's time to add motion to your shapes.

Next Get started Next class

Adding Motion

Previous Next Get started

This lesson teaches you to

1. Rotate a Shape
2. Enable Continuous Rendering

You should also read

- OpenGL

Download the sample

OpenGLES.zip

Drawing objects on screen is a pretty basic feature of OpenGL, but you can do this with other Android graphics framwork classes, including `Canvas` and `Drawable` objects. OpenGL ES provides additional capabilities for moving and transforming drawn objects in three dimensions or in other unique ways to create compelling user experiences.

In this lesson, you take another step forward into using OpenGL ES by learning how to add motion to a shape with rotation.

Rotate a Shape

Rotating a drawing object with OpenGL ES 2.0 is relatively simple. In your renderer, create another transformation matrix (a rotation matrix) and then combine it with your projection and camera view transformation matrices:

```
private float[] mRotationMatrix = new float[16];
```

```
public void onDrawFrame(GL10 gl) {
    float[] scratch = new float[16];

    ...

    // Create a rotation transformation for the triangle
    long time = SystemClock.uptimeMillis() % 4000L;
    float angle = 0.090f * ((int) time);
    Matrix.setRotateM(mRotationMatrix, 0, angle, 0, 0, -1.0f);

    // Combine the rotation matrix with the projection and camera
view
    // Note that the mMVPMatrix factor *must be first* in order
    // for the matrix multiplication product to be correct.
    Matrix.multiplyMM(scratch, 0, mMVPMatrix, 0, mRotationMatrix,
0);

    // Draw triangle
    mTriangle.draw(scratch);
}
```

If your triangle does not rotate after making these changes, make sure you have commented out the `GLSurfaceView.RENDERMODE_WHEN_DIRTY` setting, as described in the next section.

Enable Continuous Rendering

If you have diligently followed along with the example code in this class to this point, make sure you comment out the line that sets the render mode only draw when dirty, otherwise OpenGL rotates the shape only one increment and then waits for a call to `requestRender()` from the `GLSurfaceView` container:

```
public MyGLSurfaceView(Context context) {
    ...
    // Render the view only when there is a change in the drawing
data.
    // To allow the triangle to rotate automatically, this line is
commented out:
    //setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
}
```

Unless you have objects changing without any user interaction, it's usually a good idea have this flag turned on. Be ready to uncomment this code, because the next lesson makes this call applicable once again.

Next Get started Next class

Responding to Touch Events

Previous Next Get started

This lesson teaches you to

1. Setup a Touch Listener
2. Expose the Rotation Angle
3. Apply Rotation

You should also read

- OpenGL

Download the sample

OpenGLES.zip

Making objects move according to a preset program like the rotating triangle is useful for getting some attention, but what if you want to have users interact with your OpenGL ES graphics? The key to making your OpenGL ES application touch interactive is expanding your implementation of `GLSurfaceView` to override the `onTouchEvent()` to listen for touch events.

This lesson shows you how to listen for touch events to let users rotate an OpenGL ES object.

Setup a Touch Listener

In order to make your OpenGL ES application respond to touch events, you must implement the `onTouchEvent()` method in your `GLSurfaceView` class. The example implementation below shows how to listen for `MotionEvent.ACTION_MOVE` events and translate them to an angle of rotation for a shape.

```
private final float TOUCH_SCALE_FACTOR = 180.0f / 320;
private float mPreviousX;
private float mPreviousY;

@Override
public boolean onTouchEvent(MotionEvent e) {
    // MotionEvent reports input details from the touch screen
    // and other input controls. In this case, you are only
    // interested in events where the touch position changed.

    float x = e.getX();
    float y = e.getY();

    switch (e.getAction()) {
        case MotionEvent.ACTION_MOVE:

            float dx = x - mPreviousX;
            float dy = y - mPreviousY;

            // reverse direction of rotation above the mid-line
            if (y > getHeight() / 2) {
              dx = dx * -1 ;
            }

            // reverse direction of rotation to left of the mid-line
            if (x < getWidth() / 2) {
              dy = dy * -1 ;
            }

            mRenderer.setAngle(
                    mRenderer.getAngle() +
                    ((dx + dy) * TOUCH_SCALE_FACTOR));
            requestRender();
    }

    mPreviousX = x;
    mPreviousY = y;
    return true;
}
```

Notice that after calculating the rotation angle, this method calls `requestRender()` to tell the renderer that it is time to render the frame. This approach is the most efficient in this example because the frame does not need to be redrawn unless there is a change in the rotation. However, it does not have any impact on efficiency unless you also request that the renderer only redraw when the data changes using the `setRenderMode()` method, so make sure this line is uncommented in the renderer:

```
public MyGLSurfaceView(Context context) {
    ...
    // Render the view only when there is a change in the drawing
data
    setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
```

```
}
```

Expose the Rotation Angle

The example code above requires that you expose the rotation angle through your renderer by adding a public member. Since the renderer code is running on a separate thread from the main user interface thread of your application, you must declare this public variable as `volatile`. Here is the code to declare the variable and expose the getter and setter pair:

```
public class MyGLRenderer implements GLSurfaceView.Renderer {
    ...

    public volatile float mAngle;

    public float getAngle() {
        return mAngle;
    }

    public void setAngle(float angle) {
        mAngle = angle;
    }
}
```

Apply Rotation

To apply the rotation generated by touch input, comment out the code that generates an angle and add `mAngle`, which contains the touch input generated angle:

```
public void onDrawFrame(GL10 gl) {
    ...
    float[] scratch = new float[16];

    // Create a rotation for the triangle
    // long time = SystemClock.uptimeMillis() % 4000L;
    // float angle = 0.090f * ((int) time);
    Matrix.setRotateM(mRotationMatrix, 0, mAngle, 0, 0, -1.0f);

    // Combine the rotation matrix with the projection and camera
view
    // Note that the mMVPMatrix factor *must be first* in order
    // for the matrix multiplication product to be correct.
    Matrix.multiplyMM(scratch, 0, mMVPMatrix, 0, mRotationMatrix, 0);

    // Draw triangle
    mTriangle.draw(scratch);
}
```

When you have completed the steps described above, run the program and drag your finger over the screen to rotate the triangle:

Figure 1. Triangle being rotated with touch input (circle shows touch location).

Next Get started Next class

Animating Views Using Scenes and Transitions

Previous Next Get started

Dependencies and Prerequisites

- Android 4.4.2 (API level 19) or higher

You should also read

- How Android Draws Views

Try it out

- BasicTransition sample
- CustomTransition sample

Video

DevBytes: Android 4.4 Transitions

The user interface of an activity often changes in response to user input and other events. For example, an activity that contains a form where users can type search queries can hide the form when the user submits it and show a list of search results in its place.

To provide visual continuity in these situations, you can animate changes between different view hierarchies in your user interface. These animations give users feedback on their actions and help them learn how your app works.

Android includes the *transitions framework*, which enables you to easily animate changes between two view hierarchies. The framework animates the views at runtime by changing some of their property values over time. The framework includes built-in animations for common effects and lets you create custom animations and transition lifecycle callbacks.

This class teaches you to use the built-in animations in the transitions framework to animate changes between view hierarchies. This class also covers how to create custom animations.

Note: For Android versions earlier than 4.4.2 (API level 19) but greater than or equal to Android 4.0 (API level 14), use the `animateLayoutChanges` attribute to animate layouts. To learn more, see Property Animation and Animating Layout Changes.

Lessons

The Transitions Framework
: Learn the main features and components of the transitions framework.

Creating a Scene
: Learn how to create a scene to store the state of a view hierarchy.

Applying a Transition
: Learn how to apply a transition between two scenes of a view hierarchy.

Creating Custom Transitions
: Learn how to create other animation effects not included in the transitions framework.

Next Get started Next class

The Transitions Framework

Previous Next Get started

This lesson covers

1. Overview
2. Scenes
3. Transitions
4. Limitations

Animating your app's user interface provides more than just visual appeal. Animations highlight changes and provide visual cues that help users learn how your app works.

To help you animate a change between one view hierarchy and another, Android provides the transitions framework. This framework applies one or more animations to all the views in the hierarchies as it changes between them.

The framework has the following features:

Group-level animations
: Applies one or more animation effects to all of the views in a view hierarchy.

Transition-based animation
: Runs animations based on the changes between starting and ending view property values.

Built-in animations
: Includes predefined animations for common effects such as fade out or movement.

Resource file support
: Loads view

Figure 1. Visual cues using user interface animation.

hierarchies
and built-in animations from layout resource files.

Lifecycle callbacks
Defines callbacks that provide finer control over the animation and hierarchy change process.

Overview

The example in Figure 1 shows how an animation provides visual cues to help the user. As the app changes from its search entry screen to its search results screen, it fades out views that are no longer in use and fades in new views.

This animation is an example of using the transitions framework. The framework animates changes to all the views in two view hierarchies. A view hierarchy can be as simple as a single view or as complex as a `ViewGroup` containing an elaborate tree of views. The framework animates each view by changing one or more of its property values over time between the initial or *starting* view hierarchy and the final or *ending* view hierarchy.

The transitions framework works in parallel with view hierarchies and animations. The purpose of the framework is to store the state of view hierarchies, change between these hierarchies in order to modify the appearance of the device screen, and animate the change by storing and applying animation definitions.

The diagram in Figure 2 illustrates the relationship between view hierarchies, framework objects, and animations:

Figure 2. Relationships in the transitions framework.

The transitions framework provides abstractions for scenes, transitions, and transition managers. These are described in detail in the following sections. To use the framework, you create scenes for the view hierarchies in your app that you plan to change between. Next, you create a transition for each animation you want to use. To start the animation between two view hierarchies, you use a transition manager specifying the transition to use and the ending scene. This procedure is described in detail in the remaining lessons in this class.

Scenes

A scene stores the state of a view hierarchy, including all its views and their property values. A view hierarchy can be a simple view or a complex tree of views and child layouts. Storing the view hierarchy state in a scene enables you to transition into that state from another scene. The framework provides the `Scene` class to represent a scene.

The transitions framework lets you create scenes from layout resource files or from `ViewGroup` objects in your code. Creating a scene in your code is useful if you generated a view hierarchy dynamically or if you are modifying it at runtime.

In most cases, you do not create a starting scene explicitly. If you have applied a transition, the framework uses the previous ending scene as the

starting scene for any subsequent transitions. If you have not applied a transition, the framework collects information about the views from the current state of the screen.

A scene can also define its own actions that run when you make a scene change. For example, this feature is useful for cleaning up view settings after you transition to a scene.

In addition to the view hierarchy and its property values, a scene also stores a reference to the parent of the view hierarchy. This root view is called a **scene root**. Changes to the scene and animations that affect the scene occur within the scene root.

To learn how to create scenes, see Creating a Scene.

Transitions

In the transitions framework, animations create a series of frames that depict a change between the view hierarchies in the starting and ending scenes. Information about the animation is stored in a `Transition` object. To run the animation, you apply the transition using a `TransitionManager` instance. The framework can transition between two different scenes or transition to a different state for the current scene.

The framework includes a set of built-in transitions for commonly-used animation effects, such as fading and resizing views. You can also define your own custom transitions to create an animation effect using the APIs in the animations framework. The transitions framework also enables you to combine different animation effects in a transition set that contains a group of individual built-in or custom transitions.

The transition lifecycle is similar to the activity lifecycle, and it represents the transition states that the framework monitors between the start and the completion of an animation. At important lifecycle states, the framework invokes callback methods that you can implement to make adjustments to your user interface at different phases of the transition.

To learn more about transitions, see Applying a Transition and Creating

Custom Transitions.

Limitations

This section lists some known limitations of the transitions framework:

- Animations applied to a `SurfaceView` may not appear correctly. `SurfaceView` instances are updated from a non-UI thread, so the updates may be out of sync with the animations of other views.
- Some specific transition types may not produce the desired animation effect when applied to a `TextureView`.
- Classes that extend `AdapterView`, such as `ListView`, manage their child views in ways that are incompatible with the transitions framework. If you try to animate a view based on `AdapterView`, the device display may hang.
- If you try to resize a `TextView` with an animation, the text will pop to a new location before the object has completely resized. To avoid this problem, do not animate the resizing of views that contain text.

Next Get started Next class

Creating a Scene

Previous Next Get started

This lesson teaches you to

1. Create a Scene From a Layout Resource
2. Create a Scene in Your Code
3. Create Scene Actions

Scenes store the state of a view hierarchy, including all its views and their property values. The transitions framework can run animations between a starting and an ending scene. The starting scene is often determined automatically from the current state of the user interface. For the ending scene, the framework enables you to create a scene from a layout resource file or from a group of views in your code.

This lesson shows you how to create scenes in your app and how to define scene actions. The next lesson shows you how to transition between two scenes.

Note: The framework can animate changes in a single view hierarchy without using scenes, as described in Apply a Transition Without Scenes. However, understanding this lesson is essential to work with transitions.

Create a Scene From a Layout Resource

You can create a `Scene` instance directly from a layout resource file. Use this technique when the view hierarchy in the file is mostly static. The resulting scene represents the state of the view hierarchy at the time you created the `Scene` instance. If you change the view hierarchy, you have to recreate the scene. The framework creates the scene from the entire view hierarchy in the file; you can not create a scene from part of a

layout file.

To create a `Scene` instance from a layout resource file, retrieve the scene root from your layout as a `ViewGroup` instance and then call the `Scene.getSceneForLayout()` method with the scene root and the resource ID of the layout file that contains the view hierarchy for the scene.

Define Layouts for Scenes

The code snippets in the rest of this section show you how to create two different scenes with the same scene root element. The snippets also demonstrate that you can load multiple unrelated `Scene` objects without implying that they are related to each other.

The example consists of the following layout definitions:

- The main layout of an activity with a text label and a child layout.
- A relative layout for the first scene with two text fields.
- A relative layout for the second scene with the same two text fields in different order.

The example is designed so that all of the animation occurs within the child layout of the main layout for the activity. The text label in the main layout remains static.

The main layout for the activity is defined as follows:

res/layout/activity_main.xml

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/master_layout">
    <TextView
        android:id="@+id/title"
        ...
        android:text="Title"/>
    <FrameLayout
        android:id="@+id/scene_root">
        <include layout="@layout/a_scene" />
    </FrameLayout>
</LinearLayout>
```

This layout definition contains a text field and a child layout for the scene

root. The layout for the first scene is included in the main layout file. This allows the app to display it as part of the initial user interface and also to load it into a scene, since the framework can load only a whole layout file into a scene.

The layout for the first scene is defined as follows:

res/layout/a_scene.xml

```
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/scene_container"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >
    <TextView
        android:id="@+id/text_view1
        android:text="Text Line 1" />
    <TextView
        android:id="@+id/text_view2
        android:text="Text Line 2" />
</RelativeLayout>
```

The layout for the second scene contains the same two text fields (with the same IDs) placed in a different order and is defined as follows:

res/layout/another_scene.xml

```
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/scene_container"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >
    <TextView
        android:id="@+id/text_view2
        android:text="Text Line 2" />
    <TextView
        android:id="@+id/text_view1
        android:text="Text Line 1" />
</RelativeLayout>
```

Generate Scenes from Layouts

After you create definitions for the two relative layouts, you can obtain an scene for each of them. This enables you to later transition between the two UI configurations. To obtain a scene, you need a reference to the scene root and the layout resource ID.

The following code snippet shows you how to get a reference to the

scene root and create two Scene objects from the layout files:

```
Scene mAScene;
Scene mAnotherScene;

// Create the scene root for the scenes in this app
mSceneRoot = (ViewGroup) findViewById(R.id.scene_root);

// Create the scenes
mAScene = Scene.getSceneForLayout(mSceneRoot, R.layout.a_scene,
this);
mAnotherScene =
    Scene.getSceneForLayout(mSceneRoot, R.layout.another_scene,
this);
```

In the app, there are now two Scene objects based on view hierarchies. Both scenes use the scene root defined by the FrameLayout element in `res/layout/activity_main.xml`.

Create a Scene in Your Code

You can also create a Scene instance in your code from a ViewGroup object. Use this technique when you modify the view hierarchies directly in your code or when you generate them dynamically.

To create a scene from a view hierarchy in your code, use the Scene(sceneRoot, viewHierarchy) constructor. Calling this constructor is equivalent to calling the Scene.getSceneForLayout() method when you have already inflated a layout file.

The following code snippet demonstrates how to create a Scene instance from the scene root element and the view hierarchy for the scene in your code:

```
Scene mScene;

// Obtain the scene root element
mSceneRoot = (ViewGroup) mSomeLayoutElement;

// Obtain the view hierarchy to add as a child of
// the scene root when this scene is entered
mViewHierarchy = (ViewGroup) someOtherLayoutElement;

// Create a scene
mScene = new Scene(mSceneRoot, mViewHierarchy);
```

Create Scene Actions

The framework enables you to define custom scene actions that the system runs when entering or exiting a scene. In many cases, defining custom scene actions is not necessary, since the framework animates the change between scenes automatically.

Scene actions are useful for handling these cases:

- Animate views that are not in the same hierarchy. You can animate views for both the starting and ending scenes using exit and entry scene actions.
- Animate views that the transitions framework cannot animate automatically, such as `ListView` objects. For more information, see Limitations.

To provide custom scene actions, define your actions as `Runnable` objects and pass them to the `Scene.setExitAction()` or `Scene.setEnterAction()` methods. The framework calls the `setExitAction()` method on the starting scene before running the transition animation and the `setEnterAction()` method on the ending scene after running the transition animation.

Note: Do not use scene actions to pass data between views in the starting and ending scenes. For more information, see Defining Transition Lifecycle Callbacks.

Next Get started Next class

Applying a Transition

Previous Next Get started

This lesson teaches you to

1. Create a Transition
2. Apply a Transition
3. Choose Specific Target Views
4. Specify Multiple Transitions
5. Apply a Transition Without Scenes
6. Define Transition Lifecycle Callbacks

In the transitions framework, animations create a series of frames that depict a change between the view hierarchies in the starting and ending scenes. The framework represents these animations as transition objects, which contain information about an animation. To run an animation, you provide the transition to use and the ending scene to a transition manager.

This lesson teaches you run an animation between two scenes using built-in transitions to move, resize, and fade views. The next lesson shows you how to define custom transitions.

Create a Transition

In the previous lesson, you learned how to create scenes that represent the state of different view hierarchies. Once you have defined the starting scene and the ending scene you want to change between, you need to create a `Transition` object that defines an animation. The framework enables you to specify a built-in transition in a resource file and inflate it in your code or to create an instance of a built-in transition directly in your code.

Table 1. Built-in transition types.

Class	Tag	Attributes	Effect

`AutoTransition`	<autoTransition/>	-	Default transition. Fade out, move and resize, and fade in views, in that order.
`Fade`	<fade/>	`android:fadingMode="[fade_in | fade_out | fade_in_out]"`	`fade_in` fades in views `fade_out` fades out views `fade_in_out` (default) does a `fade_out` followed by a `fade_in`.
`ChangeBounds`	<changeBounds/>	-	Moves and resizes views.

Create a transition instance from a resource file

This technique enables you to modify your transition definition without having to change the code of your activity. This technique is also useful to separate complex transition definitions from your application code, as shown in Specify Multiple Transitions.

To specify a built-in transition in a resource file, follow these steps:

1. Add the `res/transition/` directory to your project.
2. Create a new XML resource file inside this directory.
3. Add an XML node for one of the built-in transitions.

For example, the following resource file specifies the `Fade` transition:

res/transition/fade_transition.xml

```
<fade xmlns:android="http://schemas.android.com/apk/res/android" />
```

The following code snippet shows how to inflate a `Transition` instance inside your activity from a resource file:

```
Transition mFadeTransition =
        TransitionInflater.from(this).
        inflateTransition(R.transition.fade_transition);
```

Create a transition instance in your code

This technique is useful for creating transition objects dynamically if you modify the user interface in your code, and to create simple built-in transition instances with few or no parameters.

To create an instance of a built-in transition, invoke one of the public constructors in the subclasses of the `Transition` class. For example, the following code snippet creates an instance of the `Fade` transition:

```
Transition mFadeTransition = new Fade();
```

Apply a Transition

You typically apply a transition to change between different view hierarchies in response to an event, such as a user action. For example, consider a search app: when the user enters a search term and clicks the search button, the app changes to the scene that represents the results layout while applying a transition that fades out the search button and fades in the search results.

To make a scene change while applying a transition in response to some event in your activity, call the `TransitionManager.go()` static method with the ending scene and the transition instance to use for the animation, as shown in the following snippet:

```
TransitionManager.go(mEndingScene, mFadeTransition);
```

The framework changes the view hierarchy inside the scene root with the view hierarchy from the ending scene while running the animation specified by the transition instance. The starting scene is the ending scene from the

last transition. If there was no previous transition, the starting scene is determined automatically from the current state of the user interface.

If you do not specify a transition instance, the transition manager can apply an automatic transition that does something reasonable for most situations. For more information, see the API reference for the `TransitionManager` class.

Choose Specific Target Views

The framework applies transitions to all views in the starting and ending scenes by default. In some cases, you may only want to apply an animation to a subset of views in a scene. For example, the framework does not support animating changes to `ListView` objects, so you should not try to animate them during a transition. The framework enables you to select specific views you want to animate.

Each view that the transition animates is called a *target*. You can only select targets that are part of the view hierarchy associated with a scene.

To remove one or more views from the list of targets, call the `removeTarget()` method before starting the transition. To add only the views you specify to the list of targets, call the `addTarget()` method. For more information, see the API reference for the `Transition` class.

Specify Multiple Transitions

To get the most impact from an animation, you should match it to the type of changes that occur between the scenes. For example, if you are removing some views and adding others between scenes, a fade out/fade in animation provides a noticeable indication that some views are no longer available. If you are moving views to different points on the screen, a better choice would be to animate the movement so that users notice the new location of the views.

You do not have to choose only one animation, since the transitions framework enables you to combine animation effects in a transition set that contains a group of individual built-in or custom transitions.

To define a transition set from a collection of transitions in XML, create a resource file in the `res/transitions/` directory and list the transitions under the `transitionSet` element. For example, the following snippet shows how to specify a transition set that has the same behaviour as the `AutoTransition` class:

```
<transitionSet
xmlns:android="http://schemas.android.com/apk/res/android"
    android:transitionOrdering="sequential">
    <fade android:fadingMode="fade_out" />
    <changeBounds />
    <fade android:fadingMode="fade_in" />
</transitionSet>
```

To inflate the transition set into a `TransitionSet` object in your code, call the `TransitionInflater.from()` method in your activity. The `TransitionSet` class extends from the `Transition` class, so you can use it with a transition manager just like any other `Transition` instance.

Apply a Transition Without Scenes

Changing view hierarchies is not the only way to modify your user interface. You can also make changes by adding, modifying, and removing child views within the current hierarchy. For example, you can implement a search interaction with just a single layout. Start with the layout showing a search entry field and a search icon. To change the user interface to show the results, remove the search button when the user clicks it by calling the `ViewGroup.removeView()` method, and add the search results by calling `ViewGroup.addView()` method.

You may want to use this approach if the alternative is to have two hierarchies that are nearly identical. Rather than having to create and maintain two separate layout files for a minor difference in the user interface, you can have one layout file containing a view hierarchy that you modify in code.

If you make changes within the current view hierarchy in this fashion, you do not need to create a scene. Instead, you can create and apply a transition between two states of a view hierarchy using a *delayed transition*. This feature of the transitions framework starts with the current view hierarchy state, records changes you make to its views, and applies a

transition that animates the changes when the system redraws the user interface.

To create a delayed transition within a single view hierarchy, follow these steps:

1. When the event that triggers the transition occurs, call the `TransitionManager.beginDelayedTransition()` method providing the parent view of all the views you want to change and the transition to use. The framework stores the current state of the child views and their property values.
2. Make changes to the child views as required by your use case. The framework records the changes you make to the child views and their properties.
3. When the system redraws the user interface according to your changes, the framework animates the changes between the original state and the new state.

The following example shows how to animate the addition of a text view to a view hierarchy using a delayed transition. The first snippet shows the layout definition file:

res/layout/activity_main.xml

```
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/mainLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >
    <EditText
        android:id="@+id/inputText"
        android:layout_alignParentLeft="true"
        android:layout_alignParentTop="true"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
    ...
</RelativeLayout>
```

The next snippet shows the code that animates the addition of the text view:

MainActivity.java

```
private TextView mLabelText;
private Fade mFade;
private ViewGroup mRootView;
```

```
...

// Load the layout
this.setContentView(R.layout.activity_main);
...

// Create a new TextView and set some View properties
mLabelText = new TextView();
mLabelText.setText("Label").setId("1");

// Get the root view and create a transition
mRootView = (ViewGroup) findViewById(R.id.mainLayout);
mFade = new Fade(IN);

// Start recording changes to the view hierarchy
TransitionManager.beginDelayedTransition(mRootView, mFade);

// Add the new TextView to the view hierarchy
mRootView.addView(mLabelText);

// When the system redraws the screen to show this update,
// the framework will animate the addition as a fade in
```

Define Transition Lifecycle Callbacks

The transition lifecycle is similar to the activity lifecycle. It represents the transition states that the framework monitors during the time between a call to the `TransitionManager.go()` method and the completion of the animation. At important lifecycle states, the framework invokes callbacks defined by the `TransitionListener` interface.

Transition lifecycle callbacks are useful, for example, for copying a view property value from the starting view hierarchy to the ending view hierarchy during a scene change. You cannot simply copy the value from its starting view to the view in the ending view hierarchy, because the ending view hierarchy is not inflated until the transition is completed. Instead, you need to store the value in a variable and then copy it into the ending view hierarchy when the framework has finished the transition. To get notified when the transition is completed, you can implement the `TransitionListener.onTransitionEnd()` method in your activity.

For more information, see the API reference for the `TransitionListener` class.

Next Get started Next class

Creating Custom Transitions

Previous Next Get started

This lesson teaches you to

1. Extend the Transition Class
2. Capture View Property Values
3. Create a Custom Animator
4. Apply a Custom Transition

A custom transition enables you to create an animation that is not available from any of the built-in transition classes. For example, you can define a custom transition that turns the foreground color of text and input fields to gray to indicate that the fields are disabled in the new screen. This type of change helps users see the fields you disabled.

A custom transition, like one of the built-in transition types, applies animations to child views of both the starting and ending scenes. Unlike built-in transition types, however, you have to provide the code that captures property values and generates animations. You may also want to define a subset of target views for your animation.

This lesson teaches you to capture property values and generate animations to create custom transitions.

Extend the Transition Class

To create a custom transition, add a class to your project that extends the `Transition` class and override the methods shown in the following snippet:

```
public class CustomTransition extends Transition {

    @Override
```

```
    public void captureStartValues(TransitionValues values) {}

    @Override
    public void captureEndValues(TransitionValues values) {}

    @Override
    public Animator createAnimator(ViewGroup sceneRoot,
                                   TransitionValues startValues,
                                   TransitionValues endValues) {}
}
```

The following sections explain how to override these methods.

Capture View Property Values

Transition animations use the property animation system described in Property Animation. Property animations change a view property between a starting and ending value over a specified period of time, so the framework needs to have both the starting and ending value of the property to construct the animation.

However, a property animation usually needs only a small subset of all the view's property values. For example, a color animation needs color property values, while a movement animation needs position property values. Since the property values needed for an animation are specific to a transition, the transitions framework does not provide every property value to a transition. Instead, the framework invokes callback methods that allow a transition to capture only the property values it needs and store them in the framework.

Capturing Starting Values

To pass the starting view values to the framework, implement the `captureStartValues(transitionValues)` method. The framework calls this method for every view in the starting scene. The method argument is a `TransitionValues` object that contains a reference to the view and a `Map` instance in which you can store the view values you want. In your implementation, retrieve these property values and pass them back to the framework by storing them in the map.

To ensure that the key for a property value does not conflict with other

TransitionValues keys, use the following naming scheme:

```
package_name:transition_name:property_name
```

The following snippet shows an implementation of the captureStartValues() method:

```
public class CustomTransition extends Transition {

    // Define a key for storing a property value in
    // TransitionValues.values with the syntax
    // package_name:transition_class:property_name to avoid
collisions
    private static final String PROPNAME_BACKGROUND =

"com.example.android.customtransition:CustomTransition:background";

    @Override
    public void captureStartValues(TransitionValues transitionValues)
{
        // Call the convenience method captureValues
        captureValues(transitionValues);
    }


    // For the view in transitionValues.view, get the values you
    // want and put them in transitionValues.values
    private void captureValues(TransitionValues transitionValues) {
        // Get a reference to the view
        View view = transitionValues.view;
        // Store its background property in the values map
        transitionValues.values.put(PROPNAME_BACKGROUND,
view.getBackground());
    }
    ...
}
```

Capture Ending Values

The framework calls the captureEndValues(TransitionValues) method once for every target view in the ending scene. In all other respects, captureEndValues() works the same as captureStartValues().

The following code snippet shows an implementation of the captureEndValues() method:

```
@Override
public void captureEndValues(TransitionValues transitionValues) {
    captureValues(transitionValues);
}
```

captureStartValues() captureEndValues()

In this example, both the and methods invoke `captureValues()` to retrieve and store values. The view property that `captureValues()` retrieves is the same, but it has different values in the starting and ending scenes. The framework maintains separate maps for the starting and ending states of a view.

Create a Custom Animator

To animate the changes to a view between its state in the starting scene and its state in the ending scene, you provide an animator by overriding the `createAnimator()` method. When the framework calls this method, it passes in the scene root view and the `TransitionValues` objects that contain the starting and ending values you captured.

The number of times the framework calls the `createAnimator()` method depends on the changes that occur between the starting and ending scenes. For example, consider a fade out/fade in animation implemented as a custom transition. If the starting scene has five targets of which two are removed from the ending scene, and the ending scene has the three targets from the starting scene plus a new target, then the framework calls `createAnimator()` six times: three of the calls animate the fading out and fading in of the targets that stay in both scene objects; two more calls animate the fading out of the targets removed from the ending scene; and one call animates the fading in of the new target in the ending scene.

For target views that exist on both the starting and ending scenes, the framework provides a `TransitionValues` object for both the `startValues` and `endValues` arguments. For target views that only exist in the starting or the ending scene, the framework provides a `TransitionValues` object for the corresponding argument and `null` for the other.

To implement the `createAnimator(ViewGroup, TransitionValues, TransitionValues)` method when you create a custom transition, use the view property values you captured to create an `Animator` object and return it to the framework. For an example implementation, see the `ChangeColor` class in the CustomTransition sample. For more information

about property animators, see Property Animation.

Apply a Custom Transition

Custom transitions work the same as built-in transitions. You can apply a custom transition using a transition manager, as described in Applying a Transition.

Next Get started Next class

Adding Animations

Previous Next Get started

Dependencies and prerequisites

- Android 4.0 or later
- Experience building an Android User Interface

You should also read

- Property Animation

Try it out

Download the sample app

Animations.zip

Animations can add subtle visual cues that notify users about what's going on in your app and improve their mental model of your app's interface. Animations are especially useful when the screen changes state, such as when content loads or new actions become available. Animations can also add a polished look to your app, which gives your app a higher quality feel.

Keep in mind though, that overusing animations or using them at the wrong time can be detrimental, such as when they cause delays. This training class shows you how to implement some common types of animations that can increase usability and add flair without annoying your users.

Lessons

Crossfading Two Views

Learn how to crossfade between two overlapping views. This lesson shows you how to crossfade a progress indicator to a view that contains text content.

Using ViewPager for Screen Slides

Learn how to animate between horizontally adjacent screens with a sliding transition.

Displaying Card Flip Animations

Learn how to animate between two views with a flipping motion.

Zooming a View

Learn how to enlarge views with a touch-to-zoom animation.

Animating Layout Changes

Learn how to enable built-in animations when adding, removing, or updating child views in a layout.

Next Get started Next class

Crossfading Two Views

Previous Next Get started

This lesson teaches you to:

1. Create the Views
2. Set up the Animation
3. Crossfade the Views

Try it out

Download the sample app

Animations.zip

Crossfade animations (also know as dissolve) gradually fade out one UI component while simultaneously fading in another. This animation is useful for situations where you want to switch content or views in your app. Crossfades are very subtle and short but offer a fluid transition from one screen to the next. When you don't use them, however, transitions often feel abrupt or hurried.

Here's an example of a crossfade from a progress indicator to some text content.

Crossfade animation

If you want to jump ahead and see a full working example, download and run the sample app and select the Crossfade example. See the following files for the code implementation:

- `src/CrossfadeActivity.java`

- `layout/activity_crossfade.xml`
- `menu/activity_crossfade.xml`

Create the Views

Create the two views that you want to crossfade. The following example creates a progress indicator and a scrollable text view:

```
<FrameLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <ScrollView
xmlns:android="http://schemas.android.com/apk/res/android"
        android:id="@+id/content"
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <TextView style="?android:textAppearanceMedium"
            android:lineSpacingMultiplier="1.2"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="@string/lorem_ipsum"
            android:padding="16dp" />

    </ScrollView>

    <ProgressBar android:id="@+id/loading_spinner"
        style="?android:progressBarStyleLarge"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center" />

</FrameLayout>
```

Set up the Animation

To set up the animation:

1. Create member variables for the views that you want to crossfade. You need these references later when modifying the views during the animation.
2. For the view that is being faded in, set its visibility to `GONE`. This prevents the view from taking up layout space and omits it from layout calculations, speeding up processing.
3. Cache the `config_shortAnimTime` system property in a member

variable. This property defines a standard "short" duration for the animation. This duration is ideal for subtle animations or animations that occur very frequently. `config_longAnimTime` and `config_mediumAnimTime` are also available if you wish to use them.

Here's an example using the layout from the previous code snippet as the activity content view:

```
public class CrossfadeActivity extends Activity {

    private View mContentView;
    private View mLoadingView;
    private int mShortAnimationDuration;

    ...

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_crossfade);

        mContentView = findViewById(R.id.content);
        mLoadingView = findViewById(R.id.loading_spinner);

        // Initially hide the content view.
        mContentView.setVisibility(View.GONE);

        // Retrieve and cache the system's default "short" animation
time.
        mShortAnimationDuration = getResources().getInteger(
                android.R.integer.config_shortAnimTime);
    }
```

Crossfade the Views

Now that the views are properly set up, crossfade them by doing the following:

1. For the view that is fading in, set the alpha value to `0` and the visibility to `VISIBLE`. (Remember that it was initially set to `GONE`.) This makes the view visible but completely transparent.
2. For the view that is fading in, animate its alpha value from `0` to `1`. At the same time, for the view that is fading out, animate the alpha value from `1` to `0`.
3. Using `onAnimationEnd()` in an `Animator.AnimatorListener`, set the visibility of the view that was fading out to `GONE`. Even though the

alpha value is `0`, setting the view's visibility to `GONE` prevents the view from taking up layout space and omits it from layout calculations, speeding up processing.

The following method shows an example of how to do this:

```
private View mContentView;
private View mLoadingView;
private int mShortAnimationDuration;

...

private void crossfade() {

    // Set the content view to 0% opacity but visible, so that it is
visible
    // (but fully transparent) during the animation.
    mContentView.setAlpha(0f);
    mContentView.setVisibility(View.VISIBLE);

    // Animate the content view to 100% opacity, and clear any
animation
    // listener set on the view.
    mContentView.animate()
            .alpha(1f)
            .setDuration(mShortAnimationDuration)
            .setListener(null);

    // Animate the loading view to 0% opacity. After the animation
ends,
    // set its visibility to GONE as an optimization step (it won't
    // participate in layout passes, etc.)
    mLoadingView.animate()
            .alpha(0f)
            .setDuration(mShortAnimationDuration)
            .setListener(new AnimatorListenerAdapter() {
                @Override
                public void onAnimationEnd(Animator animation) {
                    mLoadingView.setVisibility(View.GONE);
                }
            });
}
```

Next Get started Next class

Using ViewPager for Screen Slides

Previous Next Get started

This lesson teaches you to

1. Create the Views
2. Create the Fragment
3. Add a ViewPager
4. Customize the Animation with PageTransformer

Try it out

Download the sample app

Animations.zip

Screen slides are transitions between one entire screen to another and are common with UIs like setup wizards or slideshows. This lesson shows you how to do screen slides with a `ViewPager` provided by the support library. `ViewPager`s can animate screen slides automatically. Here's what a screen slide looks like that transitions from one screen of content to the next:

Screen slide animation

If you want to jump ahead and see a full working example, download and run the sample app and select the Screen Slide example. See the following files for the code implementation:

- `src/ScreenSlidePageFragment.java`

- `src/ScreenSlideActivity.java`
- `layout/activity_screen_slide.xml`
- `layout/fragment_screen_slide_page.xml`

Create the Views

Create a layout file that you'll later use for the content of a fragment. The following example contains a text view to display some text:

```
<!-- fragment_screen_slide_page.xml -->
<ScrollView
xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/content"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <TextView style="?android:textAppearanceMedium"
        android:padding="16dp"
        android:lineSpacingMultiplier="1.2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/lorem_ipsum" />
</ScrollView>
```

Define also a string for the contents of the fragment.

Create the Fragment

Create a `Fragment` class that returns the layout that you just created in the `onCreateView()` method. You can then create instances of this fragment in the parent activity whenever you need a new page to display to the user:

```
import android.support.v4.app.Fragment;
...
public class ScreenSlidePageFragment extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup
container,
            Bundle savedInstanceState) {
        ViewGroup rootView = (ViewGroup) inflater.inflate(
                R.layout.fragment_screen_slide_page, container,
false);

        return rootView;
    }
}
```

Add a ViewPager

ViewPagers have built-in swipe gestures to transition through pages, and they display screen slide animations by default, so you don't need to create any. ViewPagers use PagerAdapters as a supply for new pages to display, so the PagerAdapter will use the fragment class that you created earlier.

To begin, create a layout that contains a ViewPager:

```
<!-- activity_screen_slide.xml -->
<android.support.v4.view.ViewPager
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

Create an activity that does the following things:

- Sets the content view to be the layout with the ViewPager.
- Creates a class that extends the FragmentStatePagerAdapter abstract class and implements the getItem() method to supply instances of `ScreenSlidePageFragment` as new pages. The pager adapter also requires that you implement the getCount() method, which returns the amount of pages the adapter will create (five in the example).
- Hooks up the PagerAdapter to the ViewPager.
- Handles the device's back button by moving backwards in the virtual stack of fragments. If the user is already on the first page, go back on the activity back stack.

```
import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentManager;
...
public class ScreenSlidePagerActivity extends FragmentActivity {
    /**
     * The number of pages (wizard steps) to show in this demo.
     */
    private static final int NUM_PAGES = 5;

    /**
     * The pager widget, which handles animation and allows swiping
horizontally to access previous
     * and next wizard steps.
     */
    private ViewPager mPager;
```

```
    /**
     * The pager adapter, which provides the pages to the view pager
widget.
     */
    private PagerAdapter mPagerAdapter;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_screen_slide);

        // Instantiate a ViewPager and a PagerAdapter.
        mPager = (ViewPager) findViewById(R.id.pager);
        mPagerAdapter = new
ScreenSlidePagerAdapter(getSupportFragmentManager());
        mPager.setAdapter(mPagerAdapter);
    }

    @Override
    public void onBackPressed() {
        if (mPager.getCurrentItem() == 0) {
            // If the user is currently looking at the first step,
allow the system to handle the
            // Back button. This calls finish() on this activity and
pops the back stack.
            super.onBackPressed();
        } else {
            // Otherwise, select the previous step.
            mPager.setCurrentItem(mPager.getCurrentItem() - 1);
        }
    }

    /**
     * A simple pager adapter that represents 5
ScreenSlidePageFragment objects, in
     * sequence.
     */
    private class ScreenSlidePagerAdapter extends
FragmentStatePagerAdapter {
        public ScreenSlidePagerAdapter(FragmentManager fm) {
            super(fm);
        }

        @Override
        public Fragment getItem(int position) {
            return new ScreenSlidePageFragment();
        }

        @Override
        public int getCount() {
            return NUM_PAGES;
        }
    }
}
```

Customize the Animation with PageTransformer

To display a different animation from the default screen slide animation, implement the `ViewPager.PageTransformer` interface and supply it to the view pager. The interface exposes a single method, `transformPage()`. At each point in the screen's transition, this method is called once for each visible page (generally there's only one visible page) and for adjacent pages just off the screen. For example, if page three is visible and the user drags towards page four, `transformPage()` is called for pages two, three, and four at each step of the gesture.

In your implementation of `transformPage()`, you can then create custom slide animations by determining which pages need to be transformed based on the position of the page on the screen, which is obtained from the `position` parameter of the `transformPage()` method.

The `position` parameter indicates where a given page is located relative to the center of the screen. It is a dynamic property that changes as the user scrolls through the pages. When a page fills the screen, its position value is `0`. When a page is drawn just off the right side of the screen, its position value is `1`. If the user scrolls halfway between pages one and two, page one has a position of -0.5 and page two has a position of 0.5. Based on the position of the pages on the screen, you can create custom slide animations by setting page properties with methods such as `setAlpha()`, `setTranslationX()`, or `setScaleY()`.

When you have an implementation of a `PageTransformer`, call `setPageTransformer()` with your implementation to apply your custom animations. For example, if you have a `PageTransformer` named `ZoomOutPageTransformer`, you can set your custom animations like this:

```
ViewPager mPager = (ViewPager) findViewById(R.id.pager);
...
mPager.setPageTransformer(true, new ZoomOutPageTransformer());
```

See the Zoom-out page transformer and Depth page transformer sections for examples and videos of a `PageTransformer`.

Zoom-out page transformer

This page transformer shrinks and fades pages when scrolling between

adjacent pages. As a page gets closer to the center, it grows back to its normal size and fades in.

`ZoomOutPageTransformer` example

```
public class ZoomOutPageTransformer implements
ViewPager.PageTransformer {
    private static final float MIN_SCALE = 0.85f;
    private static final float MIN_ALPHA = 0.5f;

    public void transformPage(View view, float position) {
        int pageWidth = view.getWidth();
        int pageHeight = view.getHeight();

        if (position < -1) { // [-Infinity,-1)
            // This page is way off-screen to the left.
            view.setAlpha(0);

        } else if (position <= 1) { // [-1,1]
            // Modify the default slide transition to shrink the
page as well
            float scaleFactor = Math.max(MIN_SCALE, 1 -
Math.abs(position));
            float vertMargin = pageHeight * (1 - scaleFactor) / 2;
            float horzMargin = pageWidth * (1 - scaleFactor) / 2;
            if (position < 0) {
                view.setTranslationX(horzMargin - vertMargin / 2);
            } else {
                view.setTranslationX(-horzMargin + vertMargin / 2);
            }

            // Scale the page down (between MIN_SCALE and 1)
            view.setScaleX(scaleFactor);
            view.setScaleY(scaleFactor);

            // Fade the page relative to its size.
            view.setAlpha(MIN_ALPHA +
                    (scaleFactor - MIN_SCALE) /
                    (1 - MIN_SCALE) * (1 - MIN_ALPHA));

        } else { // (1,+Infinity]
            // This page is way off-screen to the right.
            view.setAlpha(0);
        }
    }
}
```

Depth page transformer

This page transformer uses the default slide animation for sliding pages to the left, while using a "depth" animation for sliding pages to the right. This depth animation fades the page out, and scales it down linearly.

`DepthPageTransformer` example

Note: During the depth animation, the default animation (a screen slide) still takes place, so you must counteract the screen slide with a negative X translation. For example:

```
view.setTranslationX(-1 * view.getWidth() * position);
```

The following example shows how to counteract the default screen slide animation in a working page transformer:

```
public class DepthPageTransformer implements
ViewPager.PageTransformer {
    private static final float MIN_SCALE = 0.75f;

    public void transformPage(View view, float position) {
        int pageWidth = view.getWidth();

        if (position < -1) { // [-Infinity,-1)
            // This page is way off-screen to the left.
            view.setAlpha(0);

        } else if (position <= 0) { // [-1,0]
            // Use the default slide transition when moving to the
left page
            view.setAlpha(1);
            view.setTranslationX(0);
            view.setScaleX(1);
            view.setScaleY(1);

        } else if (position <= 1) { // (0,1]
            // Fade the page out.
            view.setAlpha(1 - position);

            // Counteract the default slide transition
            view.setTranslationX(pageWidth * -position);

            // Scale the page down (between MIN_SCALE and 1)
            float scaleFactor = MIN_SCALE
                    + (1 - MIN_SCALE) * (1 - Math.abs(position));
            view.setScaleX(scaleFactor);
            view.setScaleY(scaleFactor);

        } else { // (1,+Infinity]
            // This page is way off-screen to the right.
            view.setAlpha(0);
        }
    }
}
```

Next Get started Next class

Displaying Card Flip Animations

Previous Next Get started

This lesson teaches you to

1. Create the Animators
2. Create the Views
3. Create the Fragment
4. Animate the Card Flip

Try it out

Download the sample app

Animations.zip

This lesson shows you how to do a card flip animation with custom fragment animations. Card flips animate between views of content by showing an animation that emulates a card flipping over.

Here's what a card flip looks like:

Card flip animation

If you want to jump ahead and see a full working example, download and run the sample app and select the Card Flip example. See the following files for the code implementation:

- `src/CardFlipActivity.java`
- `animator/card_flip_right_in.xml`
- `animator/card_flip_right_out.xml`
- `animator/card_flip_left_in.xml`

- `animator/card_flip_left_out.xml`
- `layout/fragment_card_back.xml`
- `layout/fragment_card_front.xml`

Create the Animators

Create the animations for the card flips. You'll need two animators for when the front of the card animates out and to the left and in and from the left. You'll also need two animators for when the back of the card animates in and from the right and out and to the right.

card_flip_left_in.xml

```
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- Before rotating, immediately set the alpha to 0. -->
    <objectAnimator
        android:valueFrom="1.0"
        android:valueTo="0.0"
        android:propertyName="alpha"
        android:duration="0" />

    <!-- Rotate. -->
    <objectAnimator
        android:valueFrom="-180"
        android:valueTo="0"
        android:propertyName="rotationY"

android:interpolator="@android:interpolator/accelerate_decelerate"
        android:duration="@integer/card_flip_time_full" />

    <!-- Half-way through the rotation (see startOffset), set the
alpha to 1. -->
    <objectAnimator
        android:valueFrom="0.0"
        android:valueTo="1.0"
        android:propertyName="alpha"
        android:startOffset="@integer/card_flip_time_half"
        android:duration="1" />
</set>
```

card_flip_left_out.xml

```
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- Rotate. -->
    <objectAnimator
        android:valueFrom="0"
        android:valueTo="180"
        android:propertyName="rotationY"

android:interpolator="@android:interpolator/accelerate decelerate"
```

```
        android:duration="@integer/card_flip_time_full" />

    <!-- Half-way through the rotation (see startOffset), set the
alpha to 0. -->
    <objectAnimator
        android:valueFrom="1.0"
        android:valueTo="0.0"
        android:propertyName="alpha"
        android:startOffset="@integer/card_flip_time_half"
        android:duration="1" />
</set>
```

card_flip_right_in.xml

```
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- Before rotating, immediately set the alpha to 0. -->
    <objectAnimator
        android:valueFrom="1.0"
        android:valueTo="0.0"
        android:propertyName="alpha"
        android:duration="0" />

    <!-- Rotate. -->
    <objectAnimator
        android:valueFrom="180"
        android:valueTo="0"
        android:propertyName="rotationY"

android:interpolator="@android:interpolator/accelerate_decelerate"
        android:duration="@integer/card_flip_time_full" />

    <!-- Half-way through the rotation (see startOffset), set the
alpha to 1. -->
    <objectAnimator
        android:valueFrom="0.0"
        android:valueTo="1.0"
        android:propertyName="alpha"
        android:startOffset="@integer/card_flip_time_half"
        android:duration="1" />
```

card_flip_right_out.xml

```
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- Rotate. -->
    <objectAnimator
        android:valueFrom="0"
        android:valueTo="-180"
        android:propertyName="rotationY"

android:interpolator="@android:interpolator/accelerate_decelerate"
        android:duration="@integer/card_flip_time_full" />

    <!-- Half-way through the rotation (see startOffset), set the
alpha to 0. -->
```

```
    <objectAnimator
        android:valueFrom="1.0"
        android:valueTo="0.0"
        android:propertyName="alpha"
        android:startOffset="@integer/card_flip_time_half"
        android:duration="1" />
</set>
```

Create the Views

Each side of the "card" is a separate layout that can contain any content you want, such as two screens of text, two images, or any combination of views to flip between. You'll then use the two layouts in the fragments that you'll later animate. The following layouts create one side of a card that shows text:

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:background="#a6c"
    android:padding="16dp"
    android:gravity="bottom">

    <TextView android:id="@android:id/text1"
        style="?android:textAppearanceLarge"
        android:textStyle="bold"
        android:textColor="#fff"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/card_back_title" />

    <TextView style="?android:textAppearanceSmall"
        android:textAllCaps="true"
        android:textColor="#80ffffff"
        android:textStyle="bold"
        android:lineSpacingMultiplier="1.2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/card_back_description" />

</LinearLayout>
```

and the other side of the card that displays an `ImageView`:

```
<ImageView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:src="@drawable/image1"
    android:scaleType="centerCrop"
    android:contentDescription="@string/description_image_1" />
```

Create the Fragment

Create fragment classes for the front and back of the card. These classes return the layouts that you created previously in the `onCreateView()` method of each fragment. You can then create instances of this fragment in the parent activity where you want to show the card. The following example shows nested fragment classes inside of the parent activity that uses them:

```
public class CardFlipActivity extends Activity {
    ...
    /**
     * A fragment representing the front of the card.
     */
    public class CardFrontFragment extends Fragment {
        @Override
        public View onCreateView(LayoutInflater inflater, ViewGroup
container,
                Bundle savedInstanceState) {
            return inflater.inflate(R.layout.fragment_card_front,
container, false);
        }
    }

    /**
     * A fragment representing the back of the card.
     */
    public class CardBackFragment extends Fragment {
        @Override
        public View onCreateView(LayoutInflater inflater, ViewGroup
container,
                Bundle savedInstanceState) {
            return inflater.inflate(R.layout.fragment_card_back,
container, false);
        }
    }
}
```

Animate the Card Flip

Now, you'll need to display the fragments inside of a parent activity. To do this, first create the layout for your activity. The following example creates a `FrameLayout` that you can add fragments to at runtime:

```
<FrameLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/container"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

In the activity code, set the content view to be the layout that you just created. It's also good idea to show a default fragment when the activity is created, so the following example activity shows you how to display the front of the card by default:

```
public class CardFlipActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_activity_card_flip);

        if (savedInstanceState == null) {
            getFragmentManager()
                    .beginTransaction()
                    .add(R.id.container, new CardFrontFragment())
                    .commit();
        }
    }
    ...
}
```

Now that you have the front of the card showing, you can show the back of the card with the flip animation at an appropriate time. Create a method to show the other side of the card that does the following things:

- Sets the custom animations that you created earlier for the fragment transitions.
- Replaces the currently displayed fragment with a new fragment and animates this event with the custom animations that you created.
- Adds the previously displayed fragment to the fragment back stack so when the user presses the *Back* button, the card flips back over.

```
private void flipCard() {
    if (mShowingBack) {
        getFragmentManager().popBackStack();
        return;
    }

    // Flip to the back.

    mShowingBack = true;

    // Create and commit a new fragment transaction that adds the
fragment for the back of
    // the card, uses custom animations, and is part of the fragment
manager's back stack.

    getFragmentManager()
            .beginTransaction()
```

```
            // Replace the default fragment animations with animator
resources representing
            // rotations when switching to the back of the card, as
well as animator
            // resources representing rotations when flipping back
to the front (e.g. when
            // the system Back button is pressed).
            .setCustomAnimations(
                    R.animator.card_flip_right_in,
R.animator.card_flip_right_out,
                    R.animator.card_flip_left_in,
R.animator.card_flip_left_out)

            // Replace any fragments currently in the container view
with a fragment
            // representing the next page (indicated by the just-
incremented currentPage
            // variable).
            .replace(R.id.container, new CardBackFragment())

            // Add this transaction to the back stack, allowing
users to press Back
            // to get to the front of the card.
            .addToBackStack(null)

            // Commit the transaction.
            .commit();
}
```

Next Get started Next class

Zooming a View

Previous Next Get started

This lesson teaches you to:

1. Create the Views
2. Set up the Zoom Animation
3. Zoom the View

Try it out

Download the sample app

Animations.zip

This lesson demonstrates how to do a touch-to-zoom animation, which is useful for apps such as photo galleries to animate a view from a thumbnail to a full-size image that fills the screen.

Here's what a touch-to-zoom animation looks like that expands an image thumbnail to fill the screen:

Zoom animation

If you want to jump ahead and see a full working example, download and run the sample app and select the Zoom example. See the following files for the code implementation:

- `src/TouchHighlightImageButton.java` (a simple helper class that shows a blue touch highlight when the image button is pressed)
- `src/ZoomActivity.java`
- `layout/activity_zoom.xml`

Create the Views

Create a layout file that contains the small and large version of the content that you want to zoom. The following example creates an `ImageButton` for clickable image thumbnail and an `ImageView` that displays the enlarged view of the image:

```
<FrameLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/container"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <LinearLayout android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        android:padding="16dp">

        <ImageButton
            android:id="@+id/thumb_button_1"
            android:layout_width="100dp"
            android:layout_height="75dp"
            android:layout_marginRight="1dp"
            android:src="@drawable/thumb1"
            android:scaleType="centerCrop"
            android:contentDescription="@string/description_image_1"
/>

    </LinearLayout>

    <!-- This initially-hidden ImageView will hold the
expanded/zoomed version of
         the images above. Without transformations applied, it takes
up the entire
         screen. To achieve the "zoom" animation, this view's bounds
are animated
         from the bounds of the thumbnail button above, to its final
laid-out
         bounds.
         -->

    <ImageView
        android:id="@+id/expanded_image"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:visibility="invisible"

android:contentDescription="@string/description_zoom_touch_close" />

</FrameLayout>
```

Set up the Zoom Animation

Once you apply your layout, set up the event handlers that trigger the zoom animation. The following example adds a `View.OnClickListener` to the `ImageButton` to execute the zoom animation when the user clicks the image button:

```
public class ZoomActivity extends FragmentActivity {
    // Hold a reference to the current animator,
    // so that it can be canceled mid-way.
    private Animator mCurrentAnimator;

    // The system "short" animation time duration, in milliseconds.
This
    // duration is ideal for subtle animations or animations that
occur
    // very frequently.
    private int mShortAnimationDuration;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_zoom);

        // Hook up clicks on the thumbnail views.

        final View thumb1View = findViewById(R.id.thumb_button_1);
        thumb1View.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                zoomImageFromThumb(thumb1View, R.drawable.image1);
            }
        });

        // Retrieve and cache the system's default "short" animation
time.
        mShortAnimationDuration = getResources().getInteger(
                android.R.integer.config_shortAnimTime);
    }
    ...
}
```

Zoom the View

You'll now need to animate from the normal sized view to the zoomed view when appropriate. In general, you need to animate from the bounds of the normal-sized view to the bounds of the larger-sized view. The following method shows you how to implement a zoom animation that zooms from an image thumbnail to an enlarged view by doing the following things:

1. Assign the high-res image to the hidden "zoomed-in" (enlarged)

ImageView. The following example loads a large image resource on the UI thread for simplicity. You will want to do this loading in a separate thread to prevent blocking on the UI thread and then set the bitmap on the UI thread. Ideally, the bitmap should not be larger than the screen size.

2. Calculate the starting and ending bounds for the ImageView.
3. Animate each of the four positioning and sizing properties X, Y, (SCALE_X, and SCALE_Y) simultaneously, from the starting bounds to the ending bounds. These four animations are added to an AnimatorSet so that they can be started at the same time.
4. Zoom back out by running a similar animation but in reverse when the user touches the screen when the image is zoomed in. You can do this by adding a View.OnClickListener to the ImageView. When clicked, the ImageView minimizes back down to the size of the image thumbnail and sets its visibility to GONE to hide it.

```
private void zoomImageFromThumb(final View thumbView, int
imageResId) {
    // If there's an animation in progress, cancel it
    // immediately and proceed with this one.
    if (mCurrentAnimator != null) {
        mCurrentAnimator.cancel();
    }

    // Load the high-resolution "zoomed-in" image.
    final ImageView expandedImageView = (ImageView) findViewById(
            R.id.expanded_image);
    expandedImageView.setImageResource(imageResId);

    // Calculate the starting and ending bounds for the zoomed-in
image.
    // This step involves lots of math. Yay, math.
    final Rect startBounds = new Rect();
    final Rect finalBounds = new Rect();
    final Point globalOffset = new Point();

    // The start bounds are the global visible rectangle of the
thumbnail,
    // and the final bounds are the global visible rectangle of the
container
    // view. Also set the container view's offset as the origin for
the
    // bounds, since that's the origin for the positioning animation
    // properties (X, Y).
    thumbView.getGlobalVisibleRect(startBounds);
    findViewById(R.id.container)
            .getGlobalVisibleRect(finalBounds, globalOffset);
    startBounds.offset(-globalOffset.x, -globalOffset.y);
    finalBounds.offset(-globalOffset.x, -globalOffset.y);

    // Adjust the start bounds to be the same aspect ratio as the
```

```
final
    // bounds using the "center crop" technique. This prevents
undesirable
    // stretching during the animation. Also calculate the start
scaling
    // factor (the end scaling factor is always 1.0).
    float startScale;
    if ((float) finalBounds.width() / finalBounds.height()
            > (float) startBounds.width() / startBounds.height()) {
        // Extend start bounds horizontally
        startScale = (float) startBounds.height() /
finalBounds.height();
        float startWidth = startScale * finalBounds.width();
        float deltaWidth = (startWidth - startBounds.width()) / 2;
        startBounds.left -= deltaWidth;
        startBounds.right += deltaWidth;
    } else {
        // Extend start bounds vertically
        startScale = (float) startBounds.width() /
finalBounds.width();
        float startHeight = startScale * finalBounds.height();
        float deltaHeight = (startHeight - startBounds.height()) /
2;
        startBounds.top -= deltaHeight;
        startBounds.bottom += deltaHeight;
    }

    // Hide the thumbnail and show the zoomed-in view. When the
animation
    // begins, it will position the zoomed-in view in the place of
the
    // thumbnail.
    thumbView.setAlpha(0f);
    expandedImageView.setVisibility(View.VISIBLE);

    // Set the pivot point for SCALE_X and SCALE_Y transformations
    // to the top-left corner of the zoomed-in view (the default
    // is the center of the view).
    expandedImageView.setPivotX(0f);
    expandedImageView.setPivotY(0f);

    // Construct and run the parallel animation of the four
translation and
    // scale properties (X, Y, SCALE_X, and SCALE_Y).
    AnimatorSet set = new AnimatorSet();
    set
            .play(ObjectAnimator.ofFloat(expandedImageView, View.X,
                    startBounds.left, finalBounds.left))
            .with(ObjectAnimator.ofFloat(expandedImageView, View.Y,
                    startBounds.top, finalBounds.top))
            .with(ObjectAnimator.ofFloat(expandedImageView,
View.SCALE_X,
            startScale,
1f)).with(ObjectAnimator.ofFloat(expandedImageView,
                    View.SCALE_Y, startScale, 1f));
    set.setDuration(mShortAnimationDuration);
    set.setInterpolator(new DecelerateInterpolator());
    set.addListener(new AnimatorListenerAdapter() {
        @Override
        public void onAnimationEnd(Animator animation) {
            mCurrentAnimator = null;
```

```
        }

        @Override
        public void onAnimationCancel(Animator animation) {
            mCurrentAnimator = null;
        }
    });
    set.start();
    mCurrentAnimator = set;

    // Upon clicking the zoomed-in image, it should zoom back down
    // to the original bounds and show the thumbnail instead of
    // the expanded image.
    final float startScaleFinal = startScale;
    expandedImageView.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            if (mCurrentAnimator != null) {
                mCurrentAnimator.cancel();
            }

            // Animate the four positioning/sizing properties in
parallel,
            // back to their original values.
            AnimatorSet set = new AnimatorSet();
            set.play(ObjectAnimator
                            .ofFloat(expandedImageView, View.X,
startBounds.left))
                            .with(ObjectAnimator
                                    .ofFloat(expandedImageView,
                                            View.Y,startBounds.top))
                            .with(ObjectAnimator
                                    .ofFloat(expandedImageView,
                                            View.SCALE_X,
startScaleFinal))
                            .with(ObjectAnimator
                                    .ofFloat(expandedImageView,
                                            View.SCALE_Y,
startScaleFinal));
            set.setDuration(mShortAnimationDuration);
            set.setInterpolator(new DecelerateInterpolator());
            set.addListener(new AnimatorListenerAdapter() {
                @Override
                public void onAnimationEnd(Animator animation) {
                    thumbView.setAlpha(1f);
                    expandedImageView.setVisibility(View.GONE);
                    mCurrentAnimator = null;
                }

                @Override
                public void onAnimationCancel(Animator animation) {
                    thumbView.setAlpha(1f);
                    expandedImageView.setVisibility(View.GONE);
                    mCurrentAnimator = null;
                }
            });
            set.start();
            mCurrentAnimator = set;
        }
    });
}
```

Next Get started Next class

Animating Layout Changes

Previous Next Get started

This lesson teaches you to:

1. Create the Layout
2. Add, Update, or Remove Items from the Layout

Try it out

Download the sample app

Animations.zip

A layout animation is a pre-loaded animation that the system runs each time you make a change to the layout configuration. All you need to do is set an attribute in the layout to tell the Android system to animate these layout changes, and system-default animations are carried out for you.

Tip: If you want to supply custom layout animations, create a `LayoutTransition` object and supply it to the layout with the `setLayoutTransition()` method.

Here's what a default layout animation looks like when adding items to a list:

Layout animation

If you want to jump ahead and see a full working example, download and run the sample app and select the Crossfade example. See the following files for the code implementation:

1. `src/LayoutChangesActivity.java`
2. `layout/activity_layout_changes.xml`
3. `menu/activity_layout_changes.xml`

Create the Layout

In your activity's layout XML file, set the `android:animateLayoutChanges` attribute to `true` for the layout that you want to enable animations for. For instance:

```
<LinearLayout android:id="@+id/container"
    android:animateLayoutChanges="true"
    ...
/>
```

Add, Update, or Remove Items from the Layout

Now, all you need to do is add, remove, or update items in the layout and the items are animated automatically:

```
private ViewGroup mContainerView;
...
private void addItem() {
    View newView;
    ...
    mContainerView.addView(newView, 0);
}
```

Next Get started Next class

Building Apps with Connectivity & the Cloud

These classes teach you how to connect your app to the world beyond the user's device. You'll learn how to connect to other devices in the area, connect to the Internet, backup and sync your app's data, and more.

Connecting Devices Wirelessly

Previous Next Get started

Dependencies and prerequisites

- Android 4.1 or higher

You should also read

- Wi-Fi P2P

Video

DevBytes: Network Service Discovery

Besides enabling communication with the cloud, Android's wireless APIs also enable communication with other devices on the same local network, and even devices which are not on a network, but are physically nearby. The addition of Network Service Discovery (NSD) takes this further by allowing an application to seek out a nearby device running services with which it can communicate. Integrating this functionality into your application helps you provide a wide range of features, such as playing games with users in the same room, pulling images from a networked NSD-enabled webcam, or remotely logging into other machines on the same network.

This class describes the key APIs for finding and connecting to other devices from your application. Specifically, it describes the NSD API for discovering available services and the Wi-Fi Peer-to-Peer (P2P) API for doing peer-to-peer wireless connections. This class also shows you how to use NSD and Wi-Fi P2P in combination to detect the services offered by a device and connect to the device when neither device is connected

to a network.

Lessons

Using Network Service Discovery

Learn how to broadcast services offered by your own application, discover services offered on the local network, and use NSD to determine the connection details for the service you wish to connect to.

Creating P2P Connections with Wi-Fi

Learn how to fetch a list of nearby peer devices, create an access point for legacy devices, and connect to other devices capable of Wi-Fi P2P connections.

Using Wi-Fi P2P for Service Discovery

Learn how to discover services published by nearby devices without being on the same network, using Wi-Fi P2P.

Next Get started Next class

Using Network Service Discovery

Previous Next Get started

This lesson teaches you how to

1. Register Your Service on the Network
2. Discover Services on the Network
3. Connect to Services on the Network
4. Unregister Your Service on Application Close

Try it out

Download the sample app

NsdChat.zip

Adding Network Service Discovery (NSD) to your app allows your users to identify other devices on the local network that support the services your app requests. This is useful for a variety of peer-to-peer applications such as file sharing or multi-player gaming. Android's NSD APIs simplify the effort required for you to implement such features.

This lesson shows you how to build an application that can broadcast its name and connection information to the local network and scan for information from other applications doing the same. Finally, this lesson shows you how to connect to the same application running on another device.

Register Your Service on the Network

Note: This step is optional. If you don't care about broadcasting your app's services over the local network, you can skip forward to the next

section, Discover Services on the Network.

To register your service on the local network, first create a NsdServiceInfo object. This object provides the information that other devices on the network use when they're deciding whether to connect to your service.

```
public void registerService(int port) {
    // Create the NsdServiceInfo object, and populate it.
    NsdServiceInfo serviceInfo  = new NsdServiceInfo();

    // The name is subject to change based on conflicts
    // with other services advertised on the same network.
    serviceInfo.setServiceName("NsdChat");
    serviceInfo.setServiceType("_http._tcp.");
    serviceInfo.setPort(port);
    ....
}
```

This code snippet sets the service name to "NsdChat". The name is visible to any device on the network that is using NSD to look for local services. Keep in mind that the name must be unique for any service on the network, and Android automatically handles conflict resolution. If two devices on the network both have the NsdChat application installed, one of them changes the service name automatically, to something like "NsdChat (1)".

The second parameter sets the service type, specifies which protocol and transport layer the application uses. The syntax is "_<protocol>._<transportlayer>". In the code snippet, the service uses HTTP protocol running over TCP. An application offering a printer service (for instance, a network printer) would set the service type to "_ipp._tcp".

Note: The International Assigned Numbers Authority (IANA) manages a centralized, authoritative list of service types used by service discovery protocols such as NSD and Bonjour. You can download the list from the IANA list of service names and port numbers. If you intend to use a new service type, you should reserve it by filling out the IANA Ports and Service registration form.

When setting the port for your service, avoid hardcoding it as this conflicts with other applications. For instance, assuming that your

application always uses port 1337 puts it in potential conflict with other installed applications that use the same port. Instead, use the device's next available port. Because this information is provided to other apps by a service broadcast, there's no need for the port your application uses to be known by other applications at compile-time. Instead, the applications can get this information from your service broadcast, right before connecting to your service.

If you're working with sockets, here's how you can initialize a socket to any available port simply by setting it to 0.

```
public void initializeServerSocket() {
    // Initialize a server socket on the next available port.
    mServerSocket = new ServerSocket(0);

    // Store the chosen port.
    mLocalPort =  mServerSocket.getLocalPort();
    ...
}
```

Now that you've defined the `NsdServiceInfo` object, you need to implement the `RegistrationListener` interface. This interface contains callbacks used by Android to alert your application of the success or failure of service registration and unregistration.

```
public void initializeRegistrationListener() {
    mRegistrationListener = new NsdManager.RegistrationListener() {

        @Override
        public void onServiceRegistered(NsdServiceInfo
NsdServiceInfo) {
            // Save the service name.  Android may have changed it
in order to
            // resolve a conflict, so update the name you initially
requested
            // with the name Android actually used.
            mServiceName = NsdServiceInfo.getServiceName();
        }

        @Override
        public void onRegistrationFailed(NsdServiceInfo serviceInfo,
int errorCode) {
            // Registration failed!  Put debugging code here to
determine why.
        }

        @Override
        public void onServiceUnregistered(NsdServiceInfo arg0) {
            // Service has been unregistered.  This only happens
when you call
            // NsdManager.unregisterService() and pass in this
```

```
listener.
        }

        @Override
        public void onUnregistrationFailed(NsdServiceInfo
serviceInfo, int errorCode) {
            // Unregistration failed.  Put debugging code here to
determine why.
        }
    };
}
```

Now you have all the pieces to register your service. Call the method `registerService()`.

Note that this method is asynchronous, so any code that needs to run after the service has been registered must go in the `onServiceRegistered()` method.

```
public void registerService(int port) {
    NsdServiceInfo serviceInfo  = new NsdServiceInfo();
    serviceInfo.setServiceName("NsdChat");
    serviceInfo.setServiceType("_http._tcp.");
    serviceInfo.setPort(port);

    mNsdManager = Context.getSystemService(Context.NSD_SERVICE);

    mNsdManager.registerService(
            serviceInfo, NsdManager.PROTOCOL_DNS_SD,
mRegistrationListener);
}
```

Discover Services on the Network

The network is teeming with life, from the beastly network printers to the docile network webcams, to the brutal, fiery battles of nearby tic-tac-toe players. The key to letting your application see this vibrant ecosystem of functionality is service discovery. Your application needs to listen to service broadcasts on the network to see what services are available, and filter out anything the application can't work with.

Service discovery, like service registration, has two steps: setting up a discovery listener with the relevant callbacks, and making a single asynchronous API call to `discoverServices()`.

First, instantiate an anonymous class that implements `NsdManager.DiscoveryListener`. The following snippet shows a simple

example:

```
public void initializeDiscoveryListener() {

    // Instantiate a new DiscoveryListener
    mDiscoveryListener = new NsdManager.DiscoveryListener() {

        //  Called as soon as service discovery begins.
        @Override
        public void onDiscoveryStarted(String regType) {
            Log.d(TAG, "Service discovery started");
        }

        @Override
        public void onServiceFound(NsdServiceInfo service) {
            // A service was found!  Do something with it.
            Log.d(TAG, "Service discovery success" + service);
            if (!service.getServiceType().equals(SERVICE_TYPE)) {
                // Service type is the string containing the
protocol and
                // transport layer for this service.
                Log.d(TAG, "Unknown Service Type: " +
service.getServiceType());
            } else if (service.getServiceName().equals(mServiceName))
{
                // The name of the service tells the user what
they'd be
                // connecting to. It could be "Bob's Chat App".
                Log.d(TAG, "Same machine: " + mServiceName);
            } else if (service.getServiceName().contains("NsdChat")){
                mNsdManager.resolveService(service,
mResolveListener);
            }
        }

        @Override
        public void onServiceLost(NsdServiceInfo service) {
            // When the network service is no longer available.
            // Internal bookkeeping code goes here.
            Log.e(TAG, "service lost" + service);
        }

        @Override
        public void onDiscoveryStopped(String serviceType) {
            Log.i(TAG, "Discovery stopped: " + serviceType);
        }

        @Override
        public void onStartDiscoveryFailed(String serviceType, int
errorCode) {
            Log.e(TAG, "Discovery failed: Error code:" + errorCode);
            mNsdManager.stopServiceDiscovery(this);
        }

        @Override
        public void onStopDiscoveryFailed(String serviceType, int
errorCode) {
            Log.e(TAG, "Discovery failed: Error code:" + errorCode);
            mNsdManager.stopServiceDiscovery(this);
        }
```

```
    };
}
```

The NSD API uses the methods in this interface to inform your application when discovery is started, when it fails, and when services are found and lost (lost means "is no longer available"). Notice that this snippet does several checks when a service is found.

1. The service name of the found service is compared to the service name of the local service to determine if the device just picked up its own broadcast (which is valid).
2. The service type is checked, to verify it's a type of service your application can connect to.
3. The service name is checked to verify connection to the correct application.

Checking the service name isn't always necessary, and is only relevant if you want to connect to a specific application. For instance, the application might only want to connect to instances of itself running on other devices. However, if the application wants to connect to a network printer, it's enough to see that the service type is "_ipp._tcp".

After setting up the listener, call `discoverServices()`, passing in the service type your application should look for, the discovery protocol to use, and the listener you just created.

```
    mNsdManager.discoverServices(
        SERVICE_TYPE, NsdManager.PROTOCOL_DNS_SD,
mDiscoveryListener);
```

Connect to Services on the Network

When your application finds a service on the network to connect to, it must first determine the connection information for that service, using the `resolveService()` method. Implement a `NsdManager.ResolveListener` to pass into this method, and use it to get a `NsdServiceInfo` containing the connection information.

```
public void initializeResolveListener() {
    mResolveListener = new NsdManager.ResolveListener() {
```

```
        @Override
        public void onResolveFailed(NsdServiceInfo serviceInfo, int
errorCode) {
            // Called when the resolve fails.  Use the error code to
debug.
            Log.e(TAG, "Resolve failed" + errorCode);
        }

        @Override
        public void onServiceResolved(NsdServiceInfo serviceInfo) {
            Log.e(TAG, "Resolve Succeeded. " + serviceInfo);

            if (serviceInfo.getServiceName().equals(mServiceName)) {
                Log.d(TAG, "Same IP.");
                return;
            }
            mService = serviceInfo;
            int port = mService.getPort();
            InetAddress host = mService.getHost();
        }
    };
}
```

Once the service is resolved, your application receives detailed service information including an IP address and port number. This is everything you need to create your own network connection to the service.

Unregister Your Service on Application Close

It's important to enable and disable NSD functionality as appropriate during the application's lifecycle. Unregistering your application when it closes down helps prevent other applications from thinking it's still active and attempting to connect to it. Also, service discovery is an expensive operation, and should be stopped when the parent Activity is paused, and re-enabled when the Activity is resumed. Override the lifecycle methods of your main Activity and insert code to start and stop service broadcast and discovery as appropriate.

```
//In your application's Activity

    @Override
    protected void onPause() {
        if (mNsdHelper != null) {
            mNsdHelper.tearDown();
        }
        super.onPause();
    }
```

```
    @Override
    protected void onResume() {
        super.onResume();
        if (mNsdHelper != null) {
            mNsdHelper.registerService(mConnection.getLocalPort());
            mNsdHelper.discoverServices();
        }
    }

    @Override
    protected void onDestroy() {
        mNsdHelper.tearDown();
        mConnection.tearDown();
        super.onDestroy();
    }

    // NsdHelper's tearDown method
        public void tearDown() {
        mNsdManager.unregisterService(mRegistrationListener);
        mNsdManager.stopServiceDiscovery(mDiscoveryListener);
    }
```

Next Get started Next class

Creating P2P Connections with Wi-Fi

Previous Next Get started

This lesson teaches you how to

1. Set Up Application Permissions
2. Set Up the Broadcast Receiver and Peer-to-Peer Manager
3. Initiate Peer Discovery
4. Fetch the List of Peers
5. Connect to a Peer

You should also read

- Wi-Fi Peer-to-Peer

The Wi-Fi peer-to-peer (P2P) APIs allow applications to connect to nearby devices without needing to connect to a network or hotspot (Android's Wi-Fi P2P framework complies with the Wi-Fi Direct™ certification program). Wi-Fi P2P allows your application to quickly find and interact with nearby devices, at a range beyond the capabilities of Bluetooth.

This lesson shows you how to find and connect to nearby devices using Wi-Fi P2P.

Set Up Application Permissions

In order to use Wi-Fi P2P, add the `CHANGE_WIFI_STATE`, `ACCESS_WIFI_STATE`, and `INTERNET` permissions to your manifest. Wi-Fi P2P doesn't require an internet connection, but it does use standard Java sockets, which require

the INTERNET permission. So you need the following permissions to use Wi-Fi P2P.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.android.nsdchat"
    ...

    <uses-permission
        android:required="true"
        android:name="android.permission.ACCESS_WIFI_STATE"/>
    <uses-permission
        android:required="true"
        android:name="android.permission.CHANGE_WIFI_STATE"/>
    <uses-permission
        android:required="true"
        android:name="android.permission.INTERNET"/>
    ...
```

Set Up a Broadcast Receiver and Peer-to-Peer Manager

To use Wi-Fi P2P, you need to listen for broadcast intents that tell your application when certain events have occurred. In your application, instantiate an IntentFilter and set it to listen for the following:

WIFI_P2P_STATE_CHANGED_ACTION
: Indicates whether Wi-Fi P2P is enabled

WIFI_P2P_PEERS_CHANGED_ACTION
: Indicates that the available peer list has changed.

WIFI_P2P_CONNECTION_CHANGED_ACTION
: Indicates the state of Wi-Fi P2P connectivity has changed.

WIFI_P2P_THIS_DEVICE_CHANGED_ACTION
: Indicates this device's configuration details have changed.

```
private final IntentFilter intentFilter = new IntentFilter();
...
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    //  Indicates a change in the Wi-Fi P2P status.

intentFilter.addAction(WifiP2pManager.WIFI_P2P_STATE_CHANGED_ACTION);

    // Indicates a change in the list of available peers.

intentFilter.addAction(WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION);
```

```
    // Indicates the state of Wi-Fi P2P connectivity has changed.
intentFilter.addAction(WifiP2pManager.WIFI_P2P_CONNECTION_CHANGED_ACTIO

    // Indicates this device's details have changed.
intentFilter.addAction(WifiP2pManager.WIFI_P2P_THIS_DEVICE_CHANGED_ACTI

    ...
}
```

At the end of the `onCreate()` method, get an instance of the `WifiP2pManager`, and call its `initialize()` method. This method returns a `WifiP2pManager.Channel` object, which you'll use later to connect your app to the Wi-Fi P2P framework.

```
@Override
Channel mChannel;
public void onCreate(Bundle savedInstanceState) {
    ....
    mManager = (WifiP2pManager)
getSystemService(Context.WIFI_P2P_SERVICE);
    mChannel = mManager.initialize(this, getMainLooper(), null);
}
```

Now create a new `BroadcastReceiver` class that you'll use to listen for changes to the System's Wi-Fi P2P state. In the `onReceive()` method, add a condition to handle each P2P state change listed above.

```
    @Override
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        if
(WifiP2pManager.WIFI_P2P_STATE_CHANGED_ACTION.equals(action)) {
            // Determine if Wifi P2P mode is enabled or not, alert
            // the Activity.
            int state =
intent.getIntExtra(WifiP2pManager.EXTRA_WIFI_STATE, -1);
            if (state == WifiP2pManager.WIFI_P2P_STATE_ENABLED) {
                activity.setIsWifiP2pEnabled(true);
            } else {
                activity.setIsWifiP2pEnabled(false);
            }
        } else if
(WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION.equals(action)) {

            // The peer list has changed!  We should probably do
something about
            // that.
```

```
        } else if
(WifiP2pManager.WIFI_P2P_CONNECTION_CHANGED_ACTION.equals(action)) {

            // Connection state changed!  We should probably do
something about
            // that.

        } else if
(WifiP2pManager.WIFI_P2P_THIS_DEVICE_CHANGED_ACTION.equals(action)) {
            DeviceListFragment fragment = (DeviceListFragment)
activity.getFragmentManager()
                    .findFragmentById(R.id.frag_list);
            fragment.updateThisDevice((WifiP2pDevice)
intent.getParcelableExtra(
                    WifiP2pManager.EXTRA_WIFI_P2P_DEVICE));

        }
    }
```

Finally, add code to register the intent filter and broadcast receiver when your main activity is active, and unregister them when the activity is paused. The best place to do this is the `onResume()` and `onPause()` methods.

```
    /** register the BroadcastReceiver with the intent values to be
matched */
    @Override
    public void onResume() {
        super.onResume();
        receiver = new WiFiDirectBroadcastReceiver(mManager,
mChannel, this);
        registerReceiver(receiver, intentFilter);
    }

    @Override
    public void onPause() {
        super.onPause();
        unregisterReceiver(receiver);
    }
```

Initiate Peer Discovery

To start searching for nearby devices with Wi-Fi P2P, call `discoverPeers()`. This method takes the following arguments:

- The `WifiP2pManager.Channel` you received back when you initialized the peer-to-peer mManager
- An implementation of `WifiP2pManager.ActionListener` with methods the system invokes for successful and unsuccessful discovery.

```
mManager.discoverPeers(mChannel, new WifiP2pManager.ActionListener()
{

        @Override
        public void onSuccess() {
            // Code for when the discovery initiation is successful
goes here.
            // No services have actually been discovered yet, so
this method
            // can often be left blank.  Code for peer discovery
goes in the
            // onReceive method, detailed below.
        }

        @Override
        public void onFailure(int reasonCode) {
            // Code for when the discovery initiation fails goes
here.
            // Alert the user that something went wrong.
        }
});
```

Keep in mind that this only *initiates* peer discovery. The `discoverPeers()` method starts the discovery process and then immediately returns. The system notifies you if the peer discovery process is successfully initiated by calling methods in the provided action listener. Also, discovery will remain active until a connection is initiated or a P2P group is formed.

Fetch the List of Peers

Now write the code that fetches and processes the list of peers. First implement the `WifiP2pManager.PeerListListener` interface, which provides information about the peers that Wi-Fi P2P has detected. The following code snippet illustrates this.

```
    private List peers = new ArrayList();
    ...

    private PeerListListener peerListListener = new
PeerListListener() {
        @Override
        public void onPeersAvailable(WifiP2pDeviceList peerList) {

            // Out with the old, in with the new.
            peers.clear();
            peers.addAll(peerList.getDeviceList());

            // If an AdapterView is backed by this data, notify it
            // of the change.  For instance, if you have a ListView
of available
            // peers, trigger an update.
            ((WiFiPeerListAdapter)
```

```
getListAdapter()).notifyDataSetChanged();
            if (peers.size() == 0) {
                Log.d(WiFiDirectActivity.TAG, "No devices found");
                return;
            }
        }
    }
```

Now modify your broadcast receiver's `onReceive()` method to call `requestPeers()` when an intent with the action `WIFI_P2P_PEERS_CHANGED_ACTION` is received. You need to pass this listener into the receiver somehow. One way is to send it as an argument to the broadcast receiver's constructor.

```
public void onReceive(Context context, Intent intent) {
    ...
    else if
(WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION.equals(action)) {

        // Request available peers from the wifi p2p manager. This
is an
        // asynchronous call and the calling activity is notified
with a
        // callback on PeerListListener.onPeersAvailable()
        if (mManager != null) {
            mManager.requestPeers(mChannel, peerListListener);
        }
        Log.d(WiFiDirectActivity.TAG, "P2P peers changed");
    }...
}
```

Now, an intent with the action `WIFI_P2P_PEERS_CHANGED_ACTION` intent will trigger a request for an updated peer list.

Connect to a Peer

In order to connect to a peer, create a new `WifiP2pConfig` object, and copy data into it from the `WifiP2pDevice` representing the device you want to connect to. Then call the `connect()` method.

```
    @Override
    public void connect() {
        // Picking the first device found on the network.
        WifiP2pDevice device = peers.get(0);

        WifiP2pConfig config = new WifiP2pConfig();
        config.deviceAddress = device.deviceAddress;
        config.wps.setup = WpsInfo.PBC;

        mManager.connect(mChannel, config, new ActionListener() {
```

```
            @Override
            public void onSuccess() {
                // WiFiDirectBroadcastReceiver will notify us.
Ignore for now.
            }

            @Override
            public void onFailure(int reason) {
                Toast.makeText(WiFiDirectActivity.this, "Connect
failed. Retry.",
                        Toast.LENGTH_SHORT).show();
            }
        });
    }
```

The `WifiP2pManager.ActionListener` implemented in this snippet only notifies you when the *initiation* succeeds or fails. To listen for *changes* in connection state, implement the `WifiP2pManager.ConnectionInfoListener` interface. Its `onConnectionInfoAvailable()` callback will notify you when the state of the connection changes. In cases where multiple devices are going to be connected to a single device (like a game with 3 or more players, or a chat app), one device will be designated the "group owner".

```
    @Override
    public void onConnectionInfoAvailable(final WifiP2pInfo info) {

        // InetAddress from WifiP2pInfo struct.
        InetAddress groupOwnerAddress =
info.groupOwnerAddress.getHostAddress());

        // After the group negotiation, we can determine the group
owner.
        if (info.groupFormed && info.isGroupOwner) {
            // Do whatever tasks are specific to the group owner.
            // One common case is creating a server thread and
accepting
            // incoming connections.
        } else if (info.groupFormed) {
            // The other device acts as the client. In this case,
            // you'll want to create a client thread that connects
to the group
            // owner.
        }
    }
```

Now go back to the `onReceive()` method of the broadcast receiver, and modify the section that listens for a `WIFI_P2P_CONNECTION_CHANGED_ACTION` intent. When this intent is received, call `requestConnectionInfo()`. This is an asynchronous call, so results will be received by the connection info listener you provide as a parameter.

```
        ...
```

```
        } else if
(WifiP2pManager.WIFI_P2P_CONNECTION_CHANGED_ACTION.equals(action)) {

            if (mManager == null) {
                return;
            }

            NetworkInfo networkInfo = (NetworkInfo) intent

.getParcelableExtra(WifiP2pManager.EXTRA_NETWORK_INFO);

            if (networkInfo.isConnected()) {

                // We are connected with the other device, request
connection
                // info to find group owner IP

                mManager.requestConnectionInfo(mChannel,
connectionListener);
            }
            ...
```

Next Get started Next class

Using Wi-Fi P2P for Service Discovery

Previous Next Get started

This lesson teaches you to

1. Set Up the Manifest
2. Add a Local Service
3. Discover Nearby Services

The first lesson in this class, Using Network Service Discovery, showed you how to discover services that are connected to a local network. However, using Wi-Fi Peer-to-Peer (P2P) Service Discovery allows you to discover the services of nearby devices directly, without being connected to a network. You can also advertise the services running on your device. These capabilities help you communicate between apps, even when no local network or hotspot is available.

While this set of APIs is similar in purpose to the Network Service Discovery APIs outlined in a previous lesson, implementing them in code is very different. This lesson shows you how to discover services available from other devices, using Wi-Fi P2P. The lesson assumes that you're already familiar with the Wi-Fi P2P API.

Set Up the Manifest

In order to use Wi-Fi P2P, add the `CHANGE_WIFI_STATE`, `ACCESS_WIFI_STATE`, and `INTERNET` permissions to your manifest. Even though Wi-Fi P2P doesn't require an Internet connection, it uses standard Java sockets, and using these in Android requires the requested permissions.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
```

```
    package="com.example.android.nsdchat"
    ...

    <uses-permission
        android:required="true"
        android:name="android.permission.ACCESS_WIFI_STATE"/>
    <uses-permission
        android:required="true"
        android:name="android.permission.CHANGE_WIFI_STATE"/>
    <uses-permission
        android:required="true"
        android:name="android.permission.INTERNET"/>
    ...
```

Add a Local Service

If you're providing a local service, you need to register it for service discovery. Once your local service is registered, the framework automatically responds to service discovery requests from peers.

To create a local service:

1. Create a `WifiP2pServiceInfo` object.
2. Populate it with information about your service.
3. Call `addLocalService()` to register the local service for service discovery.

```
     private void startRegistration() {
        //  Create a string map containing information about your
service.
        Map record = new HashMap();
        record.put("listenport", String.valueOf(SERVER_PORT));
        record.put("buddyname", "John Doe" + (int) (Math.random() *
1000));
        record.put("available", "visible");

        // Service information.  Pass it an instance name, service
type
        // _protocol._transportlayer , and the map containing
        // information other devices will want once they connect to
this one.
        WifiP2pDnsSdServiceInfo serviceInfo =
                WifiP2pDnsSdServiceInfo.newInstance("_test",
"_presence._tcp", record);

        // Add the local service, sending the service info, network
channel,
        // and listener that will be used to indicate success or
failure of
        // the request.
        mManager.addLocalService(channel, serviceInfo, new
ActionListener() {
```

```
            @Override
            public void onSuccess() {
                // Command successful! Code isn't necessarily needed
here,
                // Unless you want to update the UI or add logging
statements.
            }

            @Override
            public void onFailure(int arg0) {
                // Command failed.  Check for P2P_UNSUPPORTED,
ERROR, or BUSY
            }
        });
    }
```

Discover Nearby Services

Android uses callback methods to notify your application of available services, so the first thing to do is set those up. Create a `WifiP2pManager.DnsSdTxtRecordListener` to listen for incoming records. This record can optionally be broadcast by other devices. When one comes in, copy the device address and any other relevant information you want into a data structure external to the current method, so you can access it later. The following example assumes that the record contains a "buddyname" field, populated with the user's identity.

```
final HashMap<String, String> buddies = new HashMap<String,
String>();
...
private void discoverService() {
    DnsSdTxtRecordListener txtListener = new DnsSdTxtRecordListener()
{
        @Override
        /* Callback includes:
         * fullDomain: full domain name: e.g
"printer._ipp._tcp.local."
         * record: TXT record dta as a map of key/value pairs.
         * device: The device running the advertised service.
         */

        public void onDnsSdTxtRecordAvailable(
                String fullDomain, Map record, WifiP2pDevice device)
{
                Log.d(TAG, "DnsSdTxtRecord available -" +
record.toString());
                buddies.put(device.deviceAddress,
record.get("buddyname"));
            }
        };
    ...
}
```

To get the service information, create a `WifiP2pManager.DnsSdServiceResponseListener`. This receives the actual description and connection information. The previous code snippet implemented a `Map` object to pair a device address with the buddy name. The service response listener uses this to link the DNS record with the corresponding service information. Once both listeners are implemented, add them to the `WifiP2pManager` using the `setDnsSdResponseListeners()` method.

```
private void discoverService() {
...

    DnsSdServiceResponseListener servListener = new
DnsSdServiceResponseListener() {
        @Override
        public void onDnsSdServiceAvailable(String instanceName,
String registrationType,
                WifiP2pDevice resourceType) {

                // Update the device name with the human-friendly
version from
                // the DnsTxtRecord, assuming one arrived.
                resourceType.deviceName = buddies
                        .containsKey(resourceType.deviceAddress) ?
buddies
                        .get(resourceType.deviceAddress) :
resourceType.deviceName;

                // Add to the custom adapter defined specifically
for showing
                // wifi devices.
                WiFiDirectServicesList fragment =
(WiFiDirectServicesList) getFragmentManager()
                        .findFragmentById(R.id.frag_peerlist);
                WiFiDevicesAdapter adapter = ((WiFiDevicesAdapter)
fragment
                        .getListAdapter());

                adapter.add(resourceType);
                adapter.notifyDataSetChanged();
                Log.d(TAG, "onBonjourServiceAvailable " +
instanceName);
        }
    };

    mManager.setDnsSdResponseListeners(channel, servListener,
txtListener);
    ...
}
```

Now create a service request and call `addServiceRequest()`. This method also takes a listener to report success or failure.

```
        serviceRequest = WifiP2pDnsSdServiceRequest.newInstance();
        mManager.addServiceRequest(channel,
                serviceRequest,
                new ActionListener() {
                    @Override
                    public void onSuccess() {
                        // Success!
                    }

                    @Override
                    public void onFailure(int code) {
                        // Command failed.  Check for
P2P_UNSUPPORTED, ERROR, or BUSY
                    }
                });
```

Finally, make the call to `discoverServices()`.

```
        mManager.discoverServices(channel, new ActionListener() {

            @Override
            public void onSuccess() {
                // Success!
            }

            @Override
            public void onFailure(int code) {
                // Command failed.  Check for P2P_UNSUPPORTED,
ERROR, or BUSY
                if (code == WifiP2pManager.P2P_UNSUPPORTED) {
                    Log.d(TAG, "P2P isn't supported on this
device.");
                else if(...)
                    ...
            }
        });
```

If all goes well, hooray, you're done! If you encounter problems, remember that the asynchronous calls you've made take an `WifiP2pManager.ActionListener` as an argument, and this provides you with callbacks indicating success or failure. To diagnose problems, put debugging code in `onFailure()`. The error code provided by the method hints at the problem. Here are the possible error values and what they mean

`P2P_UNSUPPORTED`
: Wi-Fi P2P isn't supported on the device running the app.

`BUSY`
: The system is to busy to process the request.

`ERROR`
: The operation failed due to an internal error.

Next Get started Next class

Performing Network Operations

Previous Next Get started

Dependencies and prerequisites

- Android 1.6 (API level 4) or higher
- A device that is able to connect to mobile and Wi-Fi networks

You should also read

- Optimizing Battery Life
- Transferring Data Without Draining the Battery
- Web Apps Overview
- Transmitting Network Data Using Volley

Try it out

Download the sample

NetworkUsage.zip

This class explains the basic tasks involved in connecting to the network, monitoring the network connection (including connection changes), and giving users control over an app's network usage. It also describes how to parse and consume XML data.

This class includes a sample application that illustrates how to perform common network operations. You can download the sample (to the right) and use it as a source of reusable code for your own application.

By going through these lessons, you'll have the fundamental building blocks for creating Android applications that download content and parse

data efficiently, while minimizing network traffic.

Note: See the class Transmitting Network Data Using Volley for information on Volley, an HTTP library that makes networking for Android apps easier and faster. Volley is available through the open AOSP repository. Volley may be able to help you streamline and improve the performance of your app's network operations.

Lessons

Connecting to the Network

Learn how to connect to the network, choose an HTTP client, and perform network operations outside of the UI thread.

Managing Network Usage

Learn how to check a device's network connection, create a preferences UI for controlling network usage, and respond to connection changes.

Parsing XML Data

Learn how to parse and consume XML data.

Next Get started Next class

Connecting to the Network

Previous Next Get started

This lesson teaches you to

1. Choose an HTTP Client
2. Check the Network Connection
3. Perform Network Operations on a Separate Thread
4. Connect and Download Data
5. Convert the InputStream to a String

You should also read

- Transmitting Network Data Using Volley
- Optimizing Battery Life
- Transferring Data Without Draining the Battery
- Web Apps Overview
- Application Fundamentals

This lesson shows you how to implement a simple application that connects to the network. It explains some of the best practices you should follow in creating even the simplest network-connected app.

Note that to perform the network operations described in this lesson, your application manifest must include the following permissions:

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

Choose an HTTP Client

Most network-connected Android apps use HTTP to send and receive

data. Android includes two HTTP clients: `HttpURLConnection` and the Apache HTTP client. Both support HTTPS, streaming uploads and downloads, configurable timeouts, IPv6, and connection pooling. We recommend using `HttpURLConnection` for applications targeted at Gingerbread and higher. For more discussion of this topic, see the blog post Android's HTTP Clients.

Check the Network Connection

Before your app attempts to connect to the network, it should check to see whether a network connection is available using `getActiveNetworkInfo()` and `isConnected()`. Remember, the device may be out of range of a network, or the user may have disabled both Wi-Fi and mobile data access. For more discussion of this topic, see the lesson Managing Network Usage.

```
public void myClickHandler(View view) {
    ...
    ConnectivityManager connMgr = (ConnectivityManager)
        getSystemService(Context.CONNECTIVITY_SERVICE);
    NetworkInfo networkInfo = connMgr.getActiveNetworkInfo();
    if (networkInfo != null && networkInfo.isConnected()) {
        // fetch data
    } else {
        // display error
    }
    ...
}
```

Perform Network Operations on a Separate Thread

Network operations can involve unpredictable delays. To prevent this from causing a poor user experience, always perform network operations on a separate thread from the UI. The `AsyncTask` class provides one of the simplest ways to fire off a new task from the UI thread. For more discussion of this topic, see the blog post Multithreading For Performance.

In the following snippet, the `myClickHandler()` method invokes `new DownloadWebpageTask().execute(stringUrl)`. The `DownloadWebpageTask`

class is a subclass of AsyncTask. DownloadWebpageTask implements the following AsyncTask methods:

- doInBackground() executes the method downloadUrl(). It passes the web page URL as a parameter. The method downloadUrl() fetches and processes the web page content. When it finishes, it passes back a result string.
- onPostExecute() takes the returned string and displays it in the UI.

```
public class HttpExampleActivity extends Activity {
    private static final String DEBUG_TAG = "HttpExample";
    private EditText urlText;
    private TextView textView;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        urlText = (EditText) findViewById(R.id.myUrl);
        textView = (TextView) findViewById(R.id.myText);
    }

    // When user clicks button, calls AsyncTask.
    // Before attempting to fetch the URL, makes sure that there is
a network connection.
    public void myClickHandler(View view) {
        // Gets the URL from the UI's text field.
        String stringUrl = urlText.getText().toString();
        ConnectivityManager connMgr = (ConnectivityManager)
            getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connMgr.getActiveNetworkInfo();
        if (networkInfo != null && networkInfo.isConnected()) {
            new DownloadWebpageTask().execute(stringUrl);
        } else {
            textView.setText("No network connection available.");
        }
    }

     // Uses AsyncTask to create a task away from the main UI
thread. This task takes a
     // URL string and uses it to create an HttpUrlConnection. Once
the connection
     // has been established, the AsyncTask downloads the contents
of the webpage as
     // an InputStream. Finally, the InputStream is converted into a
string, which is
     // displayed in the UI by the AsyncTask's onPostExecute method.
     private class DownloadWebpageTask extends AsyncTask<String,
Void, String> {
        @Override
        protected String doInBackground(String... urls) {

            // params comes from the execute() call: params[0] is
the url.
            try {
                return downloadUrl(urls[0]);
```

```
            } catch (IOException e) {
                return "Unable to retrieve web page. URL may be
invalid.";
            }
        }
        // onPostExecute displays the results of the AsyncTask.
        @Override
        protected void onPostExecute(String result) {
            textView.setText(result);
        }
    }
    ...
}
```

The sequence of events in this snippet is as follows:

1. When users click the button that invokes `myClickHandler()`, the app passes the specified URL to the `AsyncTask` subclass `DownloadWebpageTask`.
2. The `AsyncTask` method `doInBackground()` calls the `downloadUrl()` method.
3. The `downloadUrl()` method takes a URL string as a parameter and uses it to create a `URL` object.
4. The `URL` object is used to establish an `HttpURLConnection`.
5. Once the connection has been established, the `HttpURLConnection` object fetches the web page content as an `InputStream`.
6. The `InputStream` is passed to the `readIt()` method, which converts the stream to a string.
7. Finally, the `AsyncTask`'s `onPostExecute()` method displays the string in the main activity's UI.

Connect and Download Data

In your thread that performs your network transactions, you can use `HttpURLConnection` to perform a `GET` and download your data. After you call `connect()`, you can get an `InputStream` of the data by calling `getInputStream()`.

In the following snippet, the `doInBackground()` method calls the method `downloadUrl()`. The `downloadUrl()` method takes the given URL and uses it to connect to the network via `HttpURLConnection`. Once a connection has been established, the app uses the method `getInputStream()` to

retrieve the data as an `InputStream`.

```
// Given a URL, establishes an HttpUrlConnection and retrieves
// the web page content as a InputStream, which it returns as
// a string.
private String downloadUrl(String myurl) throws IOException {
    InputStream is = null;
    // Only display the first 500 characters of the retrieved
    // web page content.
    int len = 500;

    try {
        URL url = new URL(myurl);
        HttpURLConnection conn = (HttpURLConnection)
url.openConnection();
        conn.setReadTimeout(10000 /* milliseconds */);
        conn.setConnectTimeout(15000 /* milliseconds */);
        conn.setRequestMethod("GET");
        conn.setDoInput(true);
        // Starts the query
        conn.connect();
        int response = conn.getResponseCode();
        Log.d(DEBUG_TAG, "The response is: " + response);
        is = conn.getInputStream();

        // Convert the InputStream into a string
        String contentAsString = readIt(is, len);
        return contentAsString;

    // Makes sure that the InputStream is closed after the app is
    // finished using it.
    } finally {
        if (is != null) {
            is.close();
        }
    }
}
```

Note that the method `getResponseCode()` returns the connection's status code. This is a useful way of getting additional information about the connection. A status code of 200 indicates success.

Convert the InputStream to a String

An `InputStream` is a readable source of bytes. Once you get an `InputStream`, it's common to decode or convert it into a target data type. For example, if you were downloading image data, you might decode and display it like this:

```
InputStream is = null;
...
Bitmap bitmap = BitmapFactory.decodeStream(is);
```

```
ImageView imageView = (ImageView) findViewById(R.id.image_view);
imageView.setImageBitmap(bitmap);
```

In the example shown above, the `InputStream` represents the text of a web page. This is how the example converts the `InputStream` to a string so that the activity can display it in the UI:

```
// Reads an InputStream and converts it to a String.
public String readIt(InputStream stream, int len) throws
IOException, UnsupportedEncodingException {
    Reader reader = null;
    reader = new InputStreamReader(stream, "UTF-8");
    char[] buffer = new char[len];
    reader.read(buffer);
    return new String(buffer);
}
```

Next Get started Next class

Managing Network Usage

Previous Next Get started

This lesson teaches you to

1. Check a Device's Network Connection
2. Manage Network Usage
3. Implement a Preferences Activity
4. Respond to Preference Changes
5. Detect Connection Changes

You should also read

- Optimizing Battery Life
- Transferring Data Without Draining the Battery
- Web Apps Overview

Try it out

Download the sample

NetworkUsage.zip

This lesson describes how to write applications that have fine-grained control over their usage of network resources. If your application performs a lot of network operations, you should provide user settings that allow users to control your app's data habits, such as how often your app syncs data, whether to perform uploads/downloads only when on Wi-Fi, whether to use data while roaming, and so on. With these controls available to them, users are much less likely to disable your app's access to background data when they approach their limits, because they can instead precisely control how much data your app uses.

For general guidelines on how to write apps that minimize the battery life impact of downloads and network connections, see Optimizing Battery Life and Transferring Data Without Draining the Battery.

Check a Device's Network Connection

A device can have various types of network connections. This lesson focuses on using either a Wi-Fi or a mobile network connection. For the full list of possible network types, see `ConnectivityManager`.

Wi-Fi is typically faster. Also, mobile data is often metered, which can get expensive. A common strategy for apps is to only fetch large data if a Wi-Fi network is available.

Before you perform network operations, it's good practice to check the state of network connectivity. Among other things, this could prevent your app from inadvertently using the wrong radio. If a network connection is unavailable, your application should respond gracefully. To check the network connection, you typically use the following classes:

- `ConnectivityManager`: Answers queries about the state of network connectivity. It also notifies applications when network connectivity changes.
- `NetworkInfo`: Describes the status of a network interface of a given type (currently either Mobile or Wi-Fi).

This code snippet tests network connectivity for Wi-Fi and mobile. It determines whether these network interfaces are available (that is, whether network connectivity is possible) and/or connected (that is, whether network connectivity exists and if it is possible to establish sockets and pass data):

```
private static final String DEBUG_TAG = "NetworkStatusExample";
...
ConnectivityManager connMgr = (ConnectivityManager)
        getSystemService(Context.CONNECTIVITY_SERVICE);
NetworkInfo networkInfo =
connMgr.getNetworkInfo(ConnectivityManager.TYPE_WIFI);
boolean isWifiConn = networkInfo.isConnected();
networkInfo =
```

```
connMgr.getNetworkInfo(ConnectivityManager.TYPE_MOBILE);
boolean isMobileConn = networkInfo.isConnected();
Log.d(DEBUG_TAG, "Wifi connected: " + isWifiConn);
Log.d(DEBUG_TAG, "Mobile connected: " + isMobileConn);
```

Note that you should not base decisions on whether a network is "available." You should always check `isConnected()` before performing network operations, since `isConnected()` handles cases like flaky mobile networks, airplane mode, and restricted background data.

A more concise way of checking whether a network interface is available is as follows. The method `getActiveNetworkInfo()` returns a `NetworkInfo` instance representing the first connected network interface it can find, or `null` if none of the interfaces is connected (meaning that an internet connection is not available):

```
public boolean isOnline() {
    ConnectivityManager connMgr = (ConnectivityManager)
            getSystemService(Context.CONNECTIVITY_SERVICE);
    NetworkInfo networkInfo = connMgr.getActiveNetworkInfo();
    return (networkInfo != null && networkInfo.isConnected());
}
```

To query more fine-grained state you can use `NetworkInfo.DetailedState`, but this should seldom be necessary.

Manage Network Usage

You can implement a preferences activity that gives users explicit control over your app's usage of network resources. For example:

- You might allow users to upload videos only when the device is connected to a Wi-Fi network.
- You might sync (or not) depending on specific criteria such as network availability, time interval, and so on.

To write an app that supports network access and managing network usage, your manifest must have the right permissions and intent filters.

- The manifest excerpted below includes the following permissions:
  - `android.permission.INTERNET`—Allows applications to open network sockets.

  - `android.permission.ACCESS_NETWORK_STATE`—Allows applications to access information about networks.
- You can declare the intent filter for the `ACTION_MANAGE_NETWORK_USAGE` action (introduced in Android 4.0) to indicate that your application defines an activity that offers options to control data usage. `ACTION_MANAGE_NETWORK_USAGE` shows settings for managing the network data usage of a specific application. When your app has a settings activity that allows users to control network usage, you should declare this intent filter for that activity. In the sample application, this action is handled by the class `SettingsActivity`, which displays a preferences UI to let users decide when to download a feed.

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

    <uses-sdk android:minSdkVersion="4"
           android:targetSdkVersion="14" />

    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission
android:name="android.permission.ACCESS_NETWORK_STATE" />

    <application
        ...>
        ...
        <activity android:label="SettingsActivity"
android:name=".SettingsActivity">
             <intent-filter>
                <action
android:name="android.intent.action.MANAGE_NETWORK_USAGE" />
                <category
android:name="android.intent.category.DEFAULT" />
          </intent-filter>
        </activity>
    </application>
</manifest>
```

Implement a Preferences Activity

As you can see in the manifest excerpt above, the sample app's activity `SettingsActivity` has an intent filter for the `ACTION_MANAGE_NETWORK_USAGE` action. `SettingsActivity` is a subclass of `PreferenceActivity`. It displays a preferences screen (shown in figure 1) that lets users specify the following:

- Whether to display summaries for each XML feed entry, or just a link for each entry.
- Whether to download the XML feed if any network connection is available, or only if Wi-Fi is available.

Figure 1. Preferences activity.

Here is `SettingsActivity`. Note that it implements `OnSharedPreferenceChangeListener`. When a user changes a preference, it fires `onSharedPreferenceChanged()`, which sets `refreshDisplay` to true. This causes the display to refresh when the user returns to the main activity:

```
public class SettingsActivity extends PreferenceActivity implements
OnSharedPreferenceChangeListener {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Loads the XML preferences file
        addPreferencesFromResource(R.xml.preferences);
    }

    @Override
```

```
    protected void onResume() {
        super.onResume();

        // Registers a listener whenever a key changes

getPreferenceScreen().getSharedPreferences().registerOnSharedPreference

    }

    @Override
    protected void onPause() {
        super.onPause();

       // Unregisters the listener set in onResume().
       // It's best practice to unregister listeners when your app
isn't using them to cut down on
       // unnecessary system overhead. You do this in onPause().

getPreferenceScreen().getSharedPreferences().unregisterOnSharedPreferen

    }

    // When the user changes the preferences selection,
    // onSharedPreferenceChanged() restarts the main activity as a
new
    // task. Sets the refreshDisplay flag to "true" to indicate that
    // the main activity should update its display.
    // The main activity queries the PreferenceManager to get the
latest settings.

    @Override
    public void onSharedPreferenceChanged(SharedPreferences
sharedPreferences, String key) {
        // Sets refreshDisplay to true so that when the user returns
to the main
        // activity, the display refreshes to reflect the new
settings.
        NetworkActivity.refreshDisplay = true;
    }
}
```

Respond to Preference Changes

When the user changes preferences in the settings screen, it typically has consequences for the app's behavior. In this snippet, the app checks the preferences settings in `onStart()`. if there is a match between the setting and the device's network connection (for example, if the setting is `"Wi-Fi"` and the device has a Wi-Fi connection), the app downloads the feed and refreshes the display.

```
public class NetworkActivity extends Activity {
    public static final String WIFI = "Wi-Fi";
    public static final String ANY = "Any";
    private static final String URL =
```

```
"http://stackoverflow.com/feeds/tag?tagnames=android&sort=newest";

    // Whether there is a Wi-Fi connection.
    private static boolean wifiConnected = false;
    // Whether there is a mobile connection.
    private static boolean mobileConnected = false;
    // Whether the display should be refreshed.
    public static boolean refreshDisplay = true;

    // The user's current network preference setting.
    public static String sPref = null;

    // The BroadcastReceiver that tracks network connectivity
changes.
    private NetworkReceiver receiver = new NetworkReceiver();

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Registers BroadcastReceiver to track network connection
changes.
        IntentFilter filter = new
IntentFilter(ConnectivityManager.CONNECTIVITY_ACTION);
        receiver = new NetworkReceiver();
        this.registerReceiver(receiver, filter);
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
        // Unregisters BroadcastReceiver when app is destroyed.
        if (receiver != null) {
            this.unregisterReceiver(receiver);
        }
    }

    // Refreshes the display if the network connection and the
    // pref settings allow it.

    @Override
    public void onStart () {
        super.onStart();

        // Gets the user's network preference settings
        SharedPreferences sharedPrefs =
PreferenceManager.getDefaultSharedPreferences(this);

        // Retrieves a string value for the preferences. The second
parameter
        // is the default value to use if a preference value is not
found.
        sPref = sharedPrefs.getString("listPref", "Wi-Fi");

        updateConnectedFlags();

        if(refreshDisplay){
            loadPage();
        }
    }
```

```
    // Checks the network connection and sets the wifiConnected and
mobileConnected
    // variables accordingly.
    public void updateConnectedFlags() {
        ConnectivityManager connMgr = (ConnectivityManager)
                getSystemService(Context.CONNECTIVITY_SERVICE);

        NetworkInfo activeInfo = connMgr.getActiveNetworkInfo();
        if (activeInfo != null && activeInfo.isConnected()) {
            wifiConnected = activeInfo.getType() ==
ConnectivityManager.TYPE_WIFI;
            mobileConnected = activeInfo.getType() ==
ConnectivityManager.TYPE_MOBILE;
        } else {
            wifiConnected = false;
            mobileConnected = false;
        }
    }

    // Uses AsyncTask subclass to download the XML feed from
stackoverflow.com.
    public void loadPage() {
        if (((sPref.equals(ANY)) && (wifiConnected ||
mobileConnected))
                || ((sPref.equals(WIFI)) && (wifiConnected))) {
            // AsyncTask subclass
            new DownloadXmlTask().execute(URL);
        } else {
            showErrorPage();
        }
    }
...

}
```

Detect Connection Changes

The final piece of the puzzle is the `BroadcastReceiver` subclass, `NetworkReceiver`. When the device's network connection changes, `NetworkReceiver` intercepts the action `CONNECTIVITY_ACTION`, determines what the network connection status is, and sets the flags `wifiConnected` and `mobileConnected` to true/false accordingly. The upshot is that the next time the user returns to the app, the app will only download the latest feed and update the display if `NetworkActivity.refreshDisplay` is set to `true`.

Setting up a BroadcastReceiver that gets called unnecessarily can be a drain on system resources. The sample application registers the `BroadcastReceiver` `NetworkReceiver` in `onCreate()`, and it unregisters it in `onDestroy()`. This is more lightweight than declaring a `<receiver>` in the

manifest. When you declare a `<receiver>` in the manifest, it can wake up your app at any time, even if you haven't run it for weeks. By registering and unregistering `NetworkReceiver` within the main activity, you ensure that the app won't be woken up after the user leaves the app. If you do declare a `<receiver>` in the manifest and you know exactly where you need it, you can use `setComponentEnabledSetting()` to enable and disable it as appropriate.

Here is `NetworkReceiver`:

```
public class NetworkReceiver extends BroadcastReceiver {

@Override
public void onReceive(Context context, Intent intent) {
    ConnectivityManager conn =  (ConnectivityManager)
        context.getSystemService(Context.CONNECTIVITY_SERVICE);
    NetworkInfo networkInfo = conn.getActiveNetworkInfo();

    // Checks the user prefs and the network connection. Based on
the result, decides whether
    // to refresh the display or keep the current display.
    // If the userpref is Wi-Fi only, checks to see if the device
has a Wi-Fi connection.
    if (WIFI.equals(sPref) && networkInfo != null &&
networkInfo.getType() == ConnectivityManager.TYPE_WIFI) {
        // If device has its Wi-Fi connection, sets refreshDisplay
        // to true. This causes the display to be refreshed when the
user
        // returns to the app.
        refreshDisplay = true;
        Toast.makeText(context, R.string.wifi_connected,
Toast.LENGTH_SHORT).show();

    // If the setting is ANY network and there is a network
connection
    // (which by process of elimination would be mobile), sets
refreshDisplay to true.
    } else if (ANY.equals(sPref) && networkInfo != null) {
        refreshDisplay = true;

    // Otherwise, the app can't download content--either because
there is no network
    // connection (mobile or Wi-Fi), or because the pref setting is
WIFI, and there
    // is no Wi-Fi connection.
    // Sets refreshDisplay to false.
    } else {
        refreshDisplay = false;
        Toast.makeText(context, R.string.lost_connection,
Toast.LENGTH_SHORT).show();
    }
}
```

Next Get started Next class

Parsing XML Data

Previous Next Get started

This lesson teaches you to

1. Choose a Parser
2. Analyze the Feed
3. Instantiate the Parser
4. Read the Feed
5. Parse XML
6. Skip Tags You Don't Care About
7. Consume XML Data

You should also read

- Web Apps Overview

Try it out

Download the sample

NetworkUsage.zip

Extensible Markup Language (XML) is a set of rules for encoding documents in machine-readable form. XML is a popular format for sharing data on the internet. Websites that frequently update their content, such as news sites or blogs, often provide an XML feed so that external programs can keep abreast of content changes. Uploading and parsing XML data is a common task for network-connected apps. This lesson explains how to parse XML documents and use their data.

Choose a Parser

We recommend `XmlPullParser`, which is an efficient and maintainable way to parse XML on Android. Historically Android has had two implementations of this interface:

- `KXmlParser` via `XmlPullParserFactory.newPullParser()`.
- `ExpatPullParser`, via `Xml.newPullParser()`.

Either choice is fine. The example in this section uses `ExpatPullParser`, via `Xml.newPullParser()`.

Analyze the Feed

The first step in parsing a feed is to decide which fields you're interested in. The parser extracts data for those fields and ignores the rest.

Here is an excerpt from the feed that's being parsed in the sample app. Each post to StackOverflow.com appears in the feed as an `entry` tag that contains several nested tags:

```
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom"
xmlns:creativeCommons="http://backend.userland.com/creativeCommonsRssMo
...">
<title type="text">newest questions tagged android - Stack
Overflow</title>
...
    <entry>
    ...
    </entry>
    <entry>
        <id>http://stackoverflow.com/q/9439999</id>
        <re:rank scheme="http://stackoverflow.com">0</re:rank>
        <title type="text">Where is my data file?</title>
        <category scheme="http://stackoverflow.com/feeds/tag?
tagnames=android&sort=newest/tags" term="android"/>
        <category scheme="http://stackoverflow.com/feeds/tag?
tagnames=android&sort=newest/tags" term="file"/>
        <author>
            <name>cliff2310</name>
            <uri>http://stackoverflow.com/users/1128925</uri>
        </author>
        <link rel="alternate"
href="http://stackoverflow.com/questions/9439999/where-is-my-data-
file" />
        <published>2012-02-25T00:30:54Z</published>
```

```
        <updated>2012-02-25T00:30:54Z</updated>
        <summary type="html">
            <p>I have an Application that requires a data
file...</p>

        </summary>
    </entry>
    <entry>
    ...
    </entry>
...
</feed>
```

The sample app extracts data for the `entry` tag and its nested tags `title`, `link`, and `summary`.

Instantiate the Parser

The next step is to instantiate a parser and kick off the parsing process. In this snippet, a parser is initialized to not process namespaces, and to use the provided `InputStream` as its input. It starts the parsing process with a call to `nextTag()` and invokes the `readFeed()` method, which extracts and processes the data the app is interested in:

```
public class StackOverflowXmlParser {
    // We don't use namespaces
    private static final String ns = null;

    public List parse(InputStream in) throws XmlPullParserException,
IOException {
        try {
            XmlPullParser parser = Xml.newPullParser();

parser.setFeature(XmlPullParser.FEATURE_PROCESS_NAMESPACES, false);
            parser.setInput(in, null);
            parser.nextTag();
            return readFeed(parser);
        } finally {
            in.close();
        }
    }
 ...
}
```

Read the Feed

The `readFeed()` method does the actual work of processing the feed. It looks for elements tagged "entry" as a starting point for recursively processing the feed. If a tag isn't an `entry` tag, it skips it. Once the whole

feed has been recursively processed, `readFeed()` returns a `List` containing the entries (including nested data members) it extracted from the feed. This `List` is then returned by the parser.

```
private List readFeed(XmlPullParser parser) throws
XmlPullParserException, IOException {
    List entries = new ArrayList();

    parser.require(XmlPullParser.START_TAG, ns, "feed");
    while (parser.next() != XmlPullParser.END_TAG) {
        if (parser.getEventType() != XmlPullParser.START_TAG) {
            continue;
        }
        String name = parser.getName();
        // Starts by looking for the entry tag
        if (name.equals("entry")) {
            entries.add(readEntry(parser));
        } else {
            skip(parser);
        }
    }
    return entries;
}
```

Parse XML

The steps for parsing an XML feed are as follows:

1. As described in Analyze the Feed, identify the tags you want to include in your app. This example extracts data for the `entry` tag and its nested tags `title`, `link`, and `summary`.
2. Create the following methods:
    - A "read" method for each tag you're interested in. For example, `readEntry()`, `readTitle()`, and so on. The parser reads tags from the input stream. When it encounters a tag named `entry`, `title`, `link` or `summary`, it calls the appropriate method for that tag. Otherwise, it skips the tag.
    - Methods to extract data for each different type of tag and to advance the parser to the next tag. For example:
        - For the `title` and `summary` tags, the parser calls `readText()`. This method extracts data for these tags by calling `parser.getText()`.
        - For the `link` tag, the parser extracts data for links by first determining if the link is the kind it's interested in. Then it

uses `parser.getAttributeValue()` to extract the link's value.
    - For the `entry` tag, the parser calls `readEntry()`. This method parses the entry's nested tags and returns an `Entry` object with the data members `title`, `link`, and `summary`.
  - A helper `skip()` method that's recursive. For more discussion of this topic, see Skip Tags You Don't Care About.

This snippet shows how the parser parses entries, titles, links, and summaries.

```
public static class Entry {
    public final String title;
    public final String link;
    public final String summary;

    private Entry(String title, String summary, String link) {
        this.title = title;
        this.summary = summary;
        this.link = link;
    }
}

// Parses the contents of an entry. If it encounters a title,
summary, or link tag, hands them off
// to their respective "read" methods for processing. Otherwise,
skips the tag.
private Entry readEntry(XmlPullParser parser) throws
XmlPullParserException, IOException {
    parser.require(XmlPullParser.START_TAG, ns, "entry");
    String title = null;
    String summary = null;
    String link = null;
    while (parser.next() != XmlPullParser.END_TAG) {
        if (parser.getEventType() != XmlPullParser.START_TAG) {
            continue;
        }
        String name = parser.getName();
        if (name.equals("title")) {
            title = readTitle(parser);
        } else if (name.equals("summary")) {
            summary = readSummary(parser);
        } else if (name.equals("link")) {
            link = readLink(parser);
        } else {
            skip(parser);
        }
    }
    return new Entry(title, summary, link);
}

// Processes title tags in the feed.
private String readTitle(XmlPullParser parser) throws IOException,
XmlPullParserException {
```

```
    parser.require(XmlPullParser.START_TAG, ns, "title");
    String title = readText(parser);
    parser.require(XmlPullParser.END_TAG, ns, "title");
    return title;
}

// Processes link tags in the feed.
private String readLink(XmlPullParser parser) throws IOException,
XmlPullParserException {
    String link = "";
    parser.require(XmlPullParser.START_TAG, ns, "link");
    String tag = parser.getName();
    String relType = parser.getAttributeValue(null, "rel");
    if (tag.equals("link")) {
        if (relType.equals("alternate")){
            link = parser.getAttributeValue(null, "href");
            parser.nextTag();
        }
    }
    parser.require(XmlPullParser.END_TAG, ns, "link");
    return link;
}

// Processes summary tags in the feed.
private String readSummary(XmlPullParser parser) throws IOException,
XmlPullParserException {
    parser.require(XmlPullParser.START_TAG, ns, "summary");
    String summary = readText(parser);
    parser.require(XmlPullParser.END_TAG, ns, "summary");
    return summary;
}

// For the tags title and summary, extracts their text values.
private String readText(XmlPullParser parser) throws IOException,
XmlPullParserException {
    String result = "";
    if (parser.next() == XmlPullParser.TEXT) {
        result = parser.getText();
        parser.nextTag();
    }
    return result;
}
  ...
}
```

Skip Tags You Don't Care About

One of the steps in the XML parsing described above is for the parser to skip tags it's not interested in. Here is the parser's `skip()` method:

```
private void skip(XmlPullParser parser) throws
XmlPullParserException, IOException {
    if (parser.getEventType() != XmlPullParser.START_TAG) {
        throw new IllegalStateException();
    }
    int depth = 1;
    while (depth != 0) {
```

```
        switch (parser.next()) {
        case XmlPullParser.END_TAG:
            depth--;
            break;
        case XmlPullParser.START_TAG:
            depth++;
            break;
        }
    }
}
```

This is how it works:

- It throws an exception if the current event isn't a `START_TAG`.
- It consumes the `START_TAG`, and all events up to and including the matching `END_TAG`.
- To make sure that it stops at the correct `END_TAG` and not at the first tag it encounters after the original `START_TAG`, it keeps track of the nesting depth.

Thus if the current element has nested elements, the value of `depth` won't be 0 until the parser has consumed all events between the original `START_TAG` and its matching `END_TAG`. For example, consider how the parser skips the `<author>` element, which has 2 nested elements, `<name>` and `<uri>`:

- The first time through the `while` loop, the next tag the parser encounters after `<author>` is the `START_TAG` for `<name>`. The value for `depth` is incremented to 2.
- The second time through the `while` loop, the next tag the parser encounters is the `END_TAG` `</name>`. The value for `depth` is decremented to 1.
- The third time through the `while` loop, the next tag the parser encounters is the `START_TAG` `<uri>`. The value for `depth` is incremented to 2.
- The fourth time through the `while` loop, the next tag the parser encounters is the `END_TAG` `</uri>`. The value for `depth` is decremented to 1.
- The fifth time and final time through the `while` loop, the next tag the parser encounters is the `END_TAG` `</author>`. The value for `depth` is decremented to 0, indicating that the `<author>` element has been successfully skipped.

Consume XML Data

The example application fetches and parses the XML feed within an AsyncTask. This takes the processing off the main UI thread. When processing is complete, the app updates the UI in the main activity (NetworkActivity).

In the excerpt shown below, the loadPage() method does the following:

- Initializes a string variable with the URL for the XML feed.
- If the user's settings and the network connection allow it, invokes new DownloadXmlTask().execute(url). This instantiates a new DownloadXmlTask object (AsyncTask subclass) and runs its execute() method, which downloads and parses the feed and returns a string result to be displayed in the UI.

```
public class NetworkActivity extends Activity {
    public static final String WIFI = "Wi-Fi";
    public static final String ANY = "Any";
    private static final String URL =
"http://stackoverflow.com/feeds/tag?tagnames=android&sort=newest";

    // Whether there is a Wi-Fi connection.
    private static boolean wifiConnected = false;
    // Whether there is a mobile connection.
    private static boolean mobileConnected = false;
    // Whether the display should be refreshed.
    public static boolean refreshDisplay = true;
    public static String sPref = null;

    ...

    // Uses AsyncTask to download the XML feed from
stackoverflow.com.
    public void loadPage() {

        if((sPref.equals(ANY)) && (wifiConnected || mobileConnected))
{
            new DownloadXmlTask().execute(URL);
        }
        else if ((sPref.equals(WIFI)) && (wifiConnected)) {
            new DownloadXmlTask().execute(URL);
        } else {
            // show error
        }
    }
```

The AsyncTask subclass shown below, DownloadXmlTask, implements the

following AsyncTask methods:

- doInBackground() executes the method `loadXmlFromNetwork()`. It passes the feed URL as a parameter. The method `loadXmlFromNetwork()` fetches and processes the feed. When it finishes, it passes back a result string.
- onPostExecute() takes the returned string and displays it in the UI.

```
// Implementation of AsyncTask used to download XML feed from
stackoverflow.com.
private class DownloadXmlTask extends AsyncTask<String, Void,
String> {
    @Override
    protected String doInBackground(String... urls) {
        try {
            return loadXmlFromNetwork(urls[0]);
        } catch (IOException e) {
            return
getResources().getString(R.string.connection_error);
        } catch (XmlPullParserException e) {
            return getResources().getString(R.string.xml_error);
        }
    }

    @Override
    protected void onPostExecute(String result) {
        setContentView(R.layout.main);
        // Displays the HTML string in the UI via a WebView
        WebView myWebView = (WebView) findViewById(R.id.webview);
        myWebView.loadData(result, "text/html", null);
    }
}
```

Below is the method `loadXmlFromNetwork()` that is invoked from `DownloadXmlTask`. It does the following:

1. Instantiates a `StackOverflowXmlParser`. It also creates variables for a List of `Entry` objects (`entries`), and `title`, `url`, and `summary`, to hold the values extracted from the XML feed for those fields.
2. Calls `downloadUrl()`, which fetches the feed and returns it as an InputStream.
3. Uses `StackOverflowXmlParser` to parse the InputStream. `StackOverflowXmlParser` populates a List of `entries` with data from the feed.
4. Processes the `entries` List, and combines the feed data with HTML markup.
5. Returns an HTML string that is displayed in the main activity UI by

the AsyncTask method onPostExecute().

```
// Uploads XML from stackoverflow.com, parses it, and combines it
with
// HTML markup. Returns HTML string.
private String loadXmlFromNetwork(String urlString) throws
XmlPullParserException, IOException {
    InputStream stream = null;
    // Instantiate the parser
    StackOverflowXmlParser stackOverflowXmlParser = new
StackOverflowXmlParser();
    List<Entry> entries = null;
    String title = null;
    String url = null;
    String summary = null;
    Calendar rightNow = Calendar.getInstance();
    DateFormat formatter = new SimpleDateFormat("MMM dd h:mmaa");

    // Checks whether the user set the preference to include summary
text
    SharedPreferences sharedPrefs =
PreferenceManager.getDefaultSharedPreferences(this);
    boolean pref = sharedPrefs.getBoolean("summaryPref", false);

    StringBuilder htmlString = new StringBuilder();
    htmlString.append("<h3>" +
getResources().getString(R.string.page_title) + "</h3>");
    htmlString.append("<em>" +
getResources().getString(R.string.updated) + " " +
            formatter.format(rightNow.getTime()) + "</em>");

    try {
        stream = downloadUrl(urlString);
        entries = stackOverflowXmlParser.parse(stream);
    // Makes sure that the InputStream is closed after the app is
    // finished using it.
    } finally {
        if (stream != null) {
            stream.close();
        }
     }

    // StackOverflowXmlParser returns a List (called "entries") of
Entry objects.
    // Each Entry object represents a single post in the XML feed.
    // This section processes the entries list to combine each entry
with HTML markup.
    // Each entry is displayed in the UI as a link that optionally
includes
    // a text summary.
    for (Entry entry : entries) {
        htmlString.append("<p><a href='");
        htmlString.append(entry.link);
        htmlString.append("'>" + entry.title + "</a></p>");
        // If the user set the preference to include summary text,
        // adds it to the display.
        if (pref) {
            htmlString.append(entry.summary);
        }
    }
```

```
    return htmlString.toString();
}

// Given a string representation of a URL, sets up a connection and
gets
// an input stream.
private InputStream downloadUrl(String urlString) throws IOException
{
    URL url = new URL(urlString);
    HttpURLConnection conn = (HttpURLConnection)
url.openConnection();
    conn.setReadTimeout(10000 /* milliseconds */);
    conn.setConnectTimeout(15000 /* milliseconds */);
    conn.setRequestMethod("GET");
    conn.setDoInput(true);
    // Starts the query
    conn.connect();
    return conn.getInputStream();
}
```

Next Get started Next class

Transferring Data Without Draining the Battery

Previous Next Get started

Dependencies and prerequisites

- Android 2.0 (API Level 5) or higher

You should also read

- Optimizing Battery Life

In this class you will learn to minimize the battery life impact of downloads and network connections, particularly in relation to the wireless radio.

This class demonstrates the best practices for scheduling and executing downloads using techniques such as caching, polling, and prefetching. You will learn how the power-use profile of the wireless radio can affect your choices on when, what, and how to transfer data in order to minimize impact on battery life.

Lessons

Optimizing Downloads for Efficient Network Access

This lesson introduces the wireless radio state machine, explains how your app's connectivity model interacts with it, and how you can minimize your data connection and use prefetching and bundling to minimize the battery drain associated with your data transfers.

Minimizing the Effect of Regular Updates

This lesson will examine how your refresh frequency can be varied

to best mitigate the effect of background updates on the underlying wireless radio state machine.

Redundant Downloads are Redundant

The most fundamental way to reduce your downloads is to download only what you need. This lesson introduces some best practices to eliminate redundant downloads.

Modifying your Download Patterns Based on the Connectivity Type

When it comes to impact on battery life, not all connection types are created equal. Not only does the Wi-Fi radio use significantly less battery than its wireless radio counterparts, but the radios used in different wireless radio technologies have different battery implications.

Next Get started Next class

Optimizing Downloads for Efficient Network Access

Previous Next Get started

This lesson teaches you to

1. Understand the radio state machine
2. Understand how apps can impact the radio state machine
3. Efficiently prefetch data
4. Batch transfers and connections
5. Reduce the number of connections you use
6. Use the DDMS Network Traffic Tool to identify areas of concern

You should also read

- Optimizing Battery Life

Using the wireless radio to transfer data is potentially one of your app's most significant sources of battery drain. To minimize the battery drain associated with network activity, it's critical that you understand how your connectivity model will affect the underlying radio hardware.

This lesson introduces the wireless radio state machine and explains how your app's connectivity model interacts with it. It goes on to propose ways to minimize your data connections, use prefetching, and bundle your transfers in order to minimize the battery drain associated with your data transfers.

The Radio State Machine

A fully active wireless radio consumes significant power, so it transitions between different energy states in order to conserve power when not in use, while attempting to minimize latency associated with "powering up" the radio when it's required.

The state machine for a typical 3G network radio consists of three energy states:

1. **Full power**: Used when a connection is active, allowing the device to transfer data at its highest possible rate.
2. **Low power**: An intermediate state that uses around 50% of the battery power at the full state.
3. **Standby**: The minimal energy state during which no network connection is active or required.

While the low and idle states drain significantly less battery, they also introduce significant latency to network requests. Returning to full power from the low state takes around 1.5

seconds, while moving from idle to full can take over 2 seconds.

To minimize latency, the state machine uses a delay to postpone the transition to lower energy states. Figure 1 uses AT&T's timings for a typical 3G radio.

Figure 1. Typical 3G wireless radio state machine.

The radio state machine on each device, particularly the associated transition delay ("tail time") and startup latency, will vary based on the wireless radio technology employed (2G, 3G, LTE, etc.) and is defined and configured by the carrier network over which the device is operating.

This lesson describes a representative state machine for a typical 3G wireless radio, based on data provided by AT&T. However, the general principles and resulting best practices are applicable for all wireless radio implementations.

This approach is particularly effective for typical web browsing as it prevents unwelcome latency while users browse the web. The relatively low tail-time also ensures that once a browsing session has finished, the radio can move to a lower energy state.

Unfortunately, this approach can lead to inefficient apps on modern smartphone OSs like Android, where apps run both in the foreground (where latency is important) and in the background (where battery life should be prioritized).

How Apps Impact the Radio State Machine

Every time you create a new network connection, the radio transitions to the full power state. In the case of the typical 3G radio state machine described above, it will remain at full power for the duration of your transfer—plus an additional 5 seconds of tail time—followed by 12 seconds at the low energy state. So for a typical 3G device, every data transfer session will cause the radio to draw energy for almost 20 seconds.

In practice, this means an app that transfers unbundled data for 1 second every 18 seconds will keep the wireless radio perpetually active, moving it back to high power just as it was about to become idle. As a result, every minute it will consume battery at the high power state for 18 seconds, and at the low power state for the remaining 42 seconds.

By comparison, the same app that bundles transfers of 3 seconds of every minute will keep the radio in the high power state for only 8 seconds, and will keep it in the low power state for only an additional 12 seconds.

The second example allows the radio to be idle for an additional 40 second every minute, resulting in a massive reduction in battery consumption.

Figure 2. Relative wireless radio power use for bundled versus unbundled transfers.

Prefetch Data

Prefetching data is an effective way to reduce the number of independent data transfer sessions. Prefetching 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.

By front loading your transfers, you reduce the number of radio activations required to download the data. As a result you not only conserve battery life, but also improve the latency, lower the required bandwidth, and reduce download times.

Prefetching also provides an improved user experience by minimizing in-app latency caused by waiting for downloads to complete before performing an action or viewing data.

However, used too aggressively, prefetching introduces the risk of increasing battery drain and bandwidth use—as well as download quota—by downloading data that isn't used. It's also important to ensure that prefetching doesn't delay application startup while the app waits for the prefetch to complete. In practical terms that might mean processing data progressively, or initiating consecutive transfers prioritized such that the data required for application startup is downloaded and processed first.

How aggressively you prefetch depends on the size of the data being downloaded and the likelihood of it being used. As a rough guide, based on the state machine described above, for data that has a 50% chance of being used within the current user session, you can typically prefetch for around 6 seconds (approximately 1-2 Mb) before the potential cost of downloading unused data matches the potential savings of not downloading that data to begin with.

Generally speaking, it's good practice to prefetch data such that you will only need to initiate another download every 2 to 5 minutes, and in the order of 1 to 5 megabytes.

Following this principle, large downloads—such as video files—should be downloaded in chunks at regular intervals (every 2 to 5 minutes), effectively prefetching only the video data likely to be viewed in the next few minutes.

Note that further downloads should be bundled, as described in the next section, Batch Transfers and Connections, and that these approximations will vary based on the connection type and speed, as discussed in Modify your Download Patterns Based on the Connectivity Type.

Let's look at some practical examples:

A music player

You could choose to prefetch an entire album, however should the user stop listening after the first song, you've wasted a significant amount of bandwidth and battery life.

A better approach would be to maintain a buffer of one song in addition to the one being played. For streaming music, rather than maintaining a continuous stream that keeps the radio active at all times, consider using HTTP live streaming to transmit the audio stream in bursts, simulating the prefetching approach described above.

A news reader

Many news apps attempt to reduce bandwidth by downloading headlines only after a category has been selected, full articles only when the user wants to read them, and thumbnails just as they scroll into view.

Using this approach, the radio will be forced to remain active for the majority of users' news-reading session as they scroll headlines, change categories, and read articles. Not only that, but the constant switching between energy states will result in significant latency when switching categories or reading articles.

A better approach would be to prefetch a reasonable amount of data at startup, beginning with the first set of news headlines and thumbnails—ensuring a low latency startup time—and continuing with the remaining headlines and thumbnails, as well as the article text for each article available from at least the primary headline list.

Another alternative is to prefetch every headline, thumbnail, article text, and possibly even full article pictures—typically in the background on a predetermined schedule. This approach risks spending significant bandwidth and battery life downloading content that's never used, so it should be implemented with caution.

One solution is to schedule the full download to occur only when connected to Wi-Fi, and possibly only when the device is charging. This is investigated in more detail in Modify your Download Patterns Based on the Connectivity Type.

Batch Transfers and Connections

Every time you initiate a connection—irrespective of the size of the associated data transfer—you potentially cause the radio to draw power for nearly 20 seconds when using a typical 3G wireless radio.

An app that pings the server every 20 seconds, just to acknowledge that the app is running and visible to the user, will keep the radio powered on indefinitely, resulting in a significant battery cost for almost no actual data transfer.

With that in mind it's important to bundle your data transfers and create a pending transfer queue. Done correctly, you can effectively phase-shift transfers that are due to occur within a similar time window, to make them all happen simultaneously—ensuring that the radio draws power for as short a duration as possible.

The underlying philosophy of this approach is to transfer as much data as possible during each transfer session in an effort to limit the number of sessions you require.

That means you should batch your transfers by queuing delay tolerant transfers, and preempting scheduled updates and prefetches, so that they are all executed when time-sensitive transfers are required. Similarly, your scheduled updates and regular prefetching should initiate the execution of your pending transfer queue.

For a practical example, let's return to the earlier examples from Prefetch Data.

Take a news application that uses the prefetching routine described above. The news reader collects analytics information to understand the reading patterns of its users and to rank the most popular stories. To keep the news fresh, it checks for updates every hour. To conserve bandwidth, rather than download full photos for each article, it prefetches only thumbnails and downloads the full photos when they are selected.

In this example, all the analytics information collected within the app should be bundled together and queued for download, rather than being transmitted as it's collected. The resulting bundle should be transferred when either a full-sized photo is being downloaded, or when an hourly update is being performed.

Any time-sensitive or on-demand transfer—such as downloading a full-sized image—should preempt regularly scheduled updates. The planned update should be executed at the same time as the on-demand transfer, with the next update scheduled to occur after the set interval. This approach mitigates the cost of performing a regular update by piggy-backing on the necessary time-sensitive photo download.

Reduce Connections

It's generally more efficient to reuse existing network connections than to initiate new ones. Reusing connections also allows the network to more intelligently react to congestion and related network data issues.

Rather than creating multiple simultaneous connections to download data, or chaining multiple consecutive GET requests, where possible you should bundle those requests into a single GET.

For example, it would be more efficient to make a single request for every news article to be returned in a single request / response than to make multiple queries for several news categories. The wireless radio needs to become active in order to transmit the termination / termination acknowledgement packets associated with server and client timeout, so it's also good practice to close your connections when they aren't in use, rather than waiting for these timeouts.

That said, closing a connection too early can prevent it from being reused, which then requires additional overhead for establishing a new connection. A useful compromise is not to close the

connection immediately, but to still close it before the inherent timeout expires.

Use the DDMS Network Traffic Tool to Identify Areas of Concern

The Android DDMS (Dalvik Debug Monitor Server) includes a Detailed Network Usage tab that makes it possible to track when your application is making network requests. Using this tool, you can monitor how and when your app transfers data and optimize the underlying code appropriately.

Figure 3 shows a pattern of transferring small amounts of data roughly 15 seconds apart, suggesting that efficiency could be dramatically improved by prefetching each request or bundling the uploads.

Figure 3. Tracking network usage with DDMS.

By monitoring the frequency of your data transfers, and the amount of data transferred during each connection, you can identify areas of your application that can be made more battery-efficient. Generally, you will be looking for short spikes that can be delayed, or that should cause a later transfer to be preempted.

To better identify the cause of transfer spikes, the Traffic Stats API allows you to tag the data transfers occurring within a thread using the `TrafficStats.setThreadStatsTag()` method, followed by manually tagging (and untagging) individual sockets using `tagSocket()` and `untagSocket()`. For example:

```
TrafficStats.setThreadStatsTag(0xF00D);
TrafficStats.tagSocket(outputSocket);
// Transfer data using socket
TrafficStats.untagSocket(outputSocket);
```

The Apache `HttpClient` and `URLConnection` libraries automatically tag sockets based on the current `getThreadStatsTag()` value. These libraries also tag and untag sockets when recycled through keep-alive pools.

```
TrafficStats.setThreadStatsTag(0xF00D);
try {
  // Make network request using HttpClient.execute()
} finally {
  TrafficStats.clearThreadStatsTag();
}
```

Socket tagging is supported in Android 4.0, but real-time stats will only be displayed on devices running Android 4.0.3 or higher.

Next Get started Next class

Minimizing the Effect of Regular Updates

Previous Next Get started

This lesson teaches you to

1. Use Google Cloud Messaging as an alternative to polling
2. Optimize polling with inexact repeating alarms and exponential back-offs

You should also read

- Optimizing Battery Life
- Google Cloud Messaging for Android

The optimal frequency of regular updates will vary based on device state, network connectivity, user behavior, and explicit user preferences.

Optimizing Battery Life discusses how to build battery-efficient apps that modify their refresh frequency based on the state of the host device. That includes disabling background service updates when you lose connectivity and reducing the rate of updates when the battery level is low.

This lesson will examine how your refresh frequency can be varied to best mitigate the effect of background updates on the underlying wireless radio state machine.

Use Google Cloud Messaging as an Alternative to Polling

Every time your app polls your server to check if an update is required, you activate the wireless radio, drawing power unnecessarily, for up to 20 seconds on a typical 3G connection.

Google Cloud Messaging for Android (GCM) is a lightweight mechanism used to transmit data from a server to a particular app instance. Using GCM, your server can notify your app running on a particular device that there is new data available for it.

Compared to polling, where your app must regularly ping the server to query for new data, this event-driven model allows your app to create a new connection only when it knows there is data to download.

The result is a reduction in unnecessary connections, and a reduced latency for updated data within your application.

GCM is implemented using a persistent TCP/IP connection. While it's possible to implement your own push service, it's best practice to use GCM. This minimizes the number of persistent connections and allows the platform to optimize bandwidth and minimize the associated impact on battery life.

Optimize Polling with Inexact Repeating Alarms and Exponential Backoffs

Where polling is required, it's good practice to set the default data refresh frequency of your app as low as possible without detracting from the user experience.

A simple approach is to offer preferences to allow users to explicitly set their required update rate, allowing them to define their own balance between data freshness and battery life.

When scheduling updates, use inexact repeating alarms that allow the system to "phase shift" the exact moment each alarm triggers.

```
int alarmType = AlarmManager.ELAPSED_REALTIME;
long interval = AlarmManager.INTERVAL_HOUR;
long start = System.currentTimeMillis() + interval;
```

```
alarmManager.setInexactRepeating(alarmType, start, interval, pi);
```

If several alarms are scheduled to trigger at similar times, this phase-shifting will cause them to be triggered simultaneously, allowing each update to piggyback on top of a single active radio state change.

Wherever possible, set your alarm type to `ELAPSED_REALTIME` or `RTC` rather than to their `_WAKEUP` equivalents. This further reduces battery impact by waiting until the phone is no longer in standby mode before the alarm triggers.

You can further reduce the impact of these scheduled alarms by opportunistically reducing their frequency based on how recently your app was used.

One approach is to implement an exponential back-off pattern to reduce the frequency of your updates (and / or the degree of prefetching you perform) if the app hasn't been used since the previous update. It's often useful to assert a minimum update frequency and to reset the frequency whenever the app is used, for example:

```
SharedPreferences sp =
  context.getSharedPreferences(PREFS, Context.MODE_WORLD_READABLE);

boolean appUsed = sp.getBoolean(PREFS_APPUSED, false);
long updateInterval = sp.getLong(PREFS_INTERVAL,
DEFAULT_REFRESH_INTERVAL);

if (!appUsed)
  if ((updateInterval *= 2) > MAX_REFRESH_INTERVAL)
    updateInterval = MAX_REFRESH_INTERVAL;

Editor spEdit = sp.edit();
spEdit.putBoolean(PREFS_APPUSED, false);
spEdit.putLong(PREFS_INTERVAL, updateInterval);
spEdit.apply();

rescheduleUpdates(updateInterval);
executeUpdateOrPrefetch();
```

You can use a similar exponential back-off pattern to reduce the effect of failed connections and download errors.

The cost of initiating a network connection is the same whether you are able to contact your server and download data or not. For time-sensitive

transfers where successful completion is important, an exponential back-off algorithm can be used to reduce the frequency of retries in order to minimize the associated battery impact, for example:

```
private void retryIn(long interval) {
  boolean success = attemptTransfer();

  if (!success) {
    retryIn(interval*2 < MAX_RETRY_INTERVAL ?
            interval*2 : MAX_RETRY_INTERVAL);
  }
}
```

Alternatively, for transfers that are failure tolerant (such as regular updates), you can simply ignore failed connection and transfer attempts.

Next Get started Next class

Redundant Downloads are Redundant

Previous Next Get started

This lesson teaches you to

1. Cache files locally
2. Use the HttpURLConnection response cache

You should also read

- Optimizing Battery Life

The most fundamental way to reduce your downloads is to download only what you need. In terms of data, that means implementing REST APIs that allow you to specify query criteria that limit the returned data by using parameters such as the time of your last update.

Similarly, when downloading images, it's good practice to reduce the size of the images server-side, rather than downloading full-sized images that are reduced on the client.

Cache Files Locally

Another important technique is to avoid downloading duplicate data. You can do this by aggressive caching. Always cache static resources, including on-demand downloads such as full size images, for as long as reasonably possible. On-demand resources should be stored separately to enable you to regularly flush your on-demand cache to manage its size.

To ensure that your caching doesn't result in your app displaying stale

data, be sure to extract the time at which the requested content was last updated, and when it expires, from within the HTTP response headers. This will allow you to determine when the associated content should be refreshed.

```
long currentTime = System.currentTimeMillis());

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

long expires = conn.getHeaderFieldDate("Expires", currentTime);
long lastModified = conn.getHeaderFieldDate("Last-Modified",
currentTime);

setDataExpirationDate(expires);

if (lastModified < lastUpdateTime) {
  // Skip update
} else {
  // Parse update
}
```

Using this approach, you can also effectively cache dynamic content while ensuring it doesn't result in your application displaying stale information.

You can cache non-sensitive data can in the unmanaged external cache directory:

```
Context.getExternalCacheDir();
```

Alternatively, you can use the managed / secure application cache. Note that this internal cache may be flushed when the system is running low on available storage.

```
Context.getCacheDir();
```

Files stored in either cache location will be erased when the application is uninstalled.

Use the HttpURLConnection Response Cache

Android 4.0 added a response cache to `HttpURLConnection`. You can enable HTTP response caching on supported devices using reflection as

follows:

```
private void enableHttpResponseCache() {
  try {
    long httpCacheSize = 10 * 1024 * 1024; // 10 MiB
    File httpCacheDir = new File(getCacheDir(), "http");
    Class.forName("android.net.http.HttpResponseCache")
         .getMethod("install", File.class, long.class)
         .invoke(null, httpCacheDir, httpCacheSize);
  } catch (Exception httpResponseCacheNotAvailable) {
    Log.d(TAG, "HTTP response cache is unavailable.");
  }
}
```

This sample code will turn on the response cache on Android 4.0+ devices without affecting earlier releases.

With the cache installed, fully cached HTTP requests can be served directly from local storage, eliminating the need to open a network connection. Conditionally cached responses can validate their freshness from the server, eliminating the bandwidth cost associated with the download.

Uncached responses get stored in the response cache for for future requests.

Next Get started Next class

Modifying your Download Patterns Based on the Connectivity Type

Previous Next Get started

This lesson teaches you to

1. Use Wi-Fi
2. Use greater bandwidth to download more data less often

You should also read

- Optimizing Battery Life

When it comes to impact on battery life, not all connection types are created equal. Not only does the Wi-Fi radio use significantly less battery than its wireless radio counterparts, but the radios used in different wireless radio technologies have different battery implications.

Use Wi-Fi

In most cases a Wi-Fi radio will offer greater bandwidth at a significantly lower battery cost. As a result, you should endeavor to perform data transfers when connected over Wi-Fi whenever possible.

You can use a broadcast receiver to listen for connectivity changes that indicate when a Wi-Fi connection has been established to execute significant downloads, preempt scheduled updates, and potentially even temporarily increase the frequency of regular updates as described in Optimizing Battery Life lesson Determining and Monitoring the

Connectivity Status.

Use Greater Bandwidth to Download More Data Less Often

When connected over a wireless radio, higher bandwidth generally comes at the price of higher battery cost. Meaning that LTE typically consumes more energy than 3G, which is in turn more expensive than 2G.

This means that while the underlying radio state machine varies based on the radio technology, generally speaking the relative battery impact of the state change tail-time is greater for higher bandwidth radios.

At the same time, the higher bandwidth means you can prefetch more aggressively, downloading more data over the same time. Perhaps less intuitively, because the tail-time battery cost is relatively higher, it's also more efficient to keep the radio active for longer periods during each transfer session to reduce the frequency of updates.

For example, if an LTE radio is has double the bandwidth and double the energy cost of 3G, you should download 4 times as much data during each session—or potentially as much as 10mb. When downloading this much data, it's important to consider the effect of your prefetching on the available local storage and flush your prefetch cache regularly.

You can use the connectivity manager to determine the active wireless radio, and modify your prefetching routines accordingly:

```
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;
}
```

Next Get started Next class

Syncing to the Cloud

Previous Next Get started

Dependencies and prerequisites

- Android 2.2 (API level 8) and higher

By providing powerful APIs for internet connectivity, the Android framework helps you build rich cloud-enabled apps that sync their data to a remote web service, making sure all your devices always stay in sync, and your valuable data is always backed up to the cloud.

This class covers strategies for backing up data using the cloud so that users can restore their data when installing your application on a new device.

Lessons

Using the Backup API

Learn how to integrate the Backup API into your Android Application, so that user data such as preferences, notes, and high scores update seamlessly across all of a user's devices

Next Get started Next class

Using the Backup API

Previous Next Get started

This lesson teaches you to

1. Register for the Android Backup Service
2. Configure Your Manifest
3. Write Your Backup Agent
4. Request a Backup
5. Restore from a Backup

You should also read

- Data Backup

When a user purchases a new device or resets their existing one, they might expect that when Google Play restores your app back to their device during the initial setup, the previous data associated with the app restores as well. By default, that doesn't happen and all the user's accomplishments or settings in your app are lost.

For situations where the volume of data is relatively light (less than a megabyte), like the user's preferences, notes, game high scores or other stats, the Backup API provides a lightweight solution. This lesson walks you through integrating the Backup API into your application, and restoring data to new devices using the Backup API.

Register for the Android Backup Service

This lesson requires the use of the Android Backup Service, which requires registration. Go ahead and register here. Once that's done, the service pre-populates an XML tag for insertion in your Android Manifest,

which looks like this:

```
<meta-data android:name="com.google.android.backup.api_key"
android:value="ABcDe1FGHij2KlmN3oPQRs4TUvW5xYZ" />
```

Note that each backup key works with a specific package name. If you have different applications, register separate keys for each one.

Configure Your Manifest

Use of the Android Backup Service requires two additions to your application manifest. First, declare the name of the class that acts as your backup agent, then add the snippet above as a child element of the Application tag. Assuming your backup agent is going to be called `TheBackupAgent`, here's an example of what the manifest looks like with this tag included:

```
<application android:label="MyApp"
             android:backupAgent="TheBackupAgent">
    ...
    <meta-data android:name="com.google.android.backup.api_key"
    android:value="ABcDe1FGHij2KlmN3oPQRs4TUvW5xYZ" />
    ...
</application>
```

Write Your Backup Agent

The easiest way to create your backup agent is by extending the wrapper class `BackupAgentHelper`. Creating this helper class is actually a very simple process. Just create a class with the same name as you used in the manifest in the previous step (in this example, `TheBackupAgent`), and extend `BackupAgentHelper`. Then override the `onCreate()`.

Inside the `onCreate()` method, create a `BackupHelper`. These helpers are specialized classes for backing up certain kinds of data. The Android framework currently includes two such helpers: `FileBackupHelper` and `SharedPreferencesBackupHelper`. After you create the helper and point it at the data you want to back up, just add it to the BackupAgentHelper using the `addHelper()` method, adding a key which is used to retrieve the data later. In most cases the entire implementation is perhaps 10 lines of

code.

Here's an example that backs up a high scores file.

```
import android.app.backup.BackupAgentHelper;
import android.app.backup.FileBackupHelper;


public class TheBackupAgent extends BackupAgentHelper {
   // The name of the SharedPreferences file
   static final String HIGH_SCORES_FILENAME = "scores";

   // A key to uniquely identify the set of backup data
   static final String FILES_BACKUP_KEY = "myfiles";

   // Allocate a helper and add it to the backup agent
   @Override
   void onCreate() {
       FileBackupHelper helper = new FileBackupHelper(this,
HIGH_SCORES_FILENAME);
       addHelper(FILES_BACKUP_KEY, helper);
   }
}
```

For added flexibility, `FileBackupHelper`'s constructor can take a variable number of filenames. You could just as easily have backed up both a high scores file and a game progress file just by adding an extra parameter, like this:

```
   @Override
   void onCreate() {
       FileBackupHelper helper = new FileBackupHelper(this,
HIGH_SCORES_FILENAME, PROGRESS_FILENAME);
       addHelper(FILES_BACKUP_KEY, helper);
   }
```

Backing up preferences is similarly easy. Create a `SharedPreferencesBackupHelper` the same way you did a `FileBackupHelper`. In this case, instead of adding filenames to the constructor, add the names of the shared preference groups being used by your application. Here's an example of how your backup agent helper might look if high scores are implemented as preferences instead of a flat file:

```
import android.app.backup.BackupAgentHelper;
import android.app.backup.SharedPreferencesBackupHelper;

public class TheBackupAgent extends BackupAgentHelper {
    // The names of the SharedPreferences groups that the
application maintains.  These
    // are the same strings that are passed to
```

```
getSharedPreferences(String, int).
     static final String PREFS_DISPLAY = "displayprefs";
     static final String PREFS_SCORES = "highscores";

     // An arbitrary string used within the BackupAgentHelper
implementation to
     // identify the SharedPreferencesBackupHelper's data.
     static final String MY_PREFS_BACKUP_KEY = "myprefs";

     // Simply allocate a helper and install it
     void onCreate() {
         SharedPreferencesBackupHelper helper =
                 new SharedPreferencesBackupHelper(this,
PREFS_DISPLAY, PREFS_SCORES);
         addHelper(MY_PREFS_BACKUP_KEY, helper);
     }
 }
```

You can add as many backup helper instances to your backup agent helper as you like, but remember that you only need one of each type. One `FileBackupHelper` handles all the files that you need to back up, and one `SharedPreferencesBackupHelper` handles all the shared preferencegroups you need backed up.

Request a Backup

In order to request a backup, just create an instance of the `BackupManager`, and call it's `dataChanged()` method.

```
import android.app.backup.BackupManager;
...

public void requestBackup() {
  BackupManager bm = new BackupManager(this);
  bm.dataChanged();
}
```

This call notifies the backup manager that there is data ready to be backed up to the cloud. At some point in the future, the backup manager then calls your backup agent's `onBackup()` method. You can make the call whenever your data has changed, without having to worry about causing excessive network activity. If you request a backup twice before a backup occurs, the backup only occurs once.

Restore from a Backup

Typically you shouldn't ever have to manually request a restore, as it happens automatically when your application is installed on a device. However, if it *is* necessary to trigger a manual restore, just call the `requestRestore()` method.

Next Get started Next class

Resolving Cloud Save Conflicts

In this document

1. Get Notified of Conflicts
2. Handle the Simple Cases
3. Design a Strategy for More Complex Cases
    1. First Attempt: Store Only the Total
    2. Second Attempt: Store the Total and the Delta
    3. Solution: Store the Sub-totals per Device
4. Clean Up Your Data

You should also read

- Cloud Save
- Cloud Save in Android

This article describes how to design a robust conflict resolution strategy for apps that save data to the cloud using the Cloud Save service. The Cloud Save service allows you to store application data for each user of an application on Google's servers. Your application can retrieve and update this user data from Android devices, iOS devices, or web applications by using the Cloud Save APIs.

Saving and loading progress in Cloud Save is straightforward: it's just a matter of serializing the player's data to and from byte arrays and storing those arrays in the cloud. However, when your user has multiple devices and two or more of them attempt to save data to the cloud, the saves might conflict, and you must decide how to resolve it. The structure of your cloud save data largely dictates how robust your conflict resolution can be, so you must design your data carefully in order to allow your conflict resolution logic to handle each case correctly.

The article starts by describing a few flawed approaches and explains

where they fall short. Then it presents a solution for avoiding conflicts. The discussion focuses on games, but you can apply the same principles to any app that saves data to the cloud.

Get Notified of Conflicts

The `OnStateLoadedListener` methods are responsible for loading an application's state data from Google's servers. The callback `OnStateLoadedListener.onStateConflict` provides a mechanism for your application to resolve conflicts between the local state on a user's device and the state stored in the cloud:

```
@Override
public void onStateConflict(int stateKey, String resolvedVersion,
    byte[] localData, byte[] serverData) {
    // resolve conflict, then call mAppStateClient.resolveConflict()
 ...
}
```

At this point your application must choose which one of the data sets should be kept, or it can submit a new data set that represents the merged data. It is up to you to implement this conflict resolution logic.

It's important to realize that the Cloud Save service synchronizes data in the background. Therefore, you should ensure that your app is prepared to receive that callback outside of the context where you originally generated the data. Specifically, if the Google Play services application detects a conflict in the background, the callback will be called the next time you attempt to load the data, which might not happen until the next time the user starts the app.

Therefore, design of your cloud save data and conflict resolution code must be *context-independent*: given two conflicting save states, you must be able to resolve the conflict using only the data available within the data sets, without consulting any external context.

Handle the Simple Cases

Here are some simple cases of conflict resolution. For many apps, it is sufficient to adopt a variant of one of these strategies:

- **New is better than old**. In some cases, new data should always replace old data. For example, if the data represents the player's choice for a character's shirt color, then a more recent choice should override an older choice. In this case, you would probably choose to store the timestamp in the cloud save data. When resolving the conflict, pick the data set with the most recent timestamp (remember to use a reliable clock, and be careful about time zone differences).
- **One set of data is clearly better than the other**. In other cases, it will always be clear which data can be defined as "best". For example, if the data represents the player's best time in a racing game, then it's clear that, in case of conflicts, you should keep the best (smallest) time.
- **Merge by union**. It may be possible to resolve the conflict by computing a union of the two conflicting sets. For example, if your data represents the set of levels that player has unlocked, then the resolved data is simply the union of the two conflicting sets. This way, players won't lose any levels they have unlocked. The CollectAllTheStars sample game uses a variant of this strategy.

Design a Strategy for More Complex Cases

A more complicated case happens when your game allows the player to collect fungible items or units, such as gold coins or experience points. Let's consider a hypothetical game, called Coin Run, an infinite runner where the goal is to collect coins and become very, very rich. Each coin collected gets added to the player's piggy bank.

The following sections describe three strategies for resolving sync conflicts between multiple devices: two that sound good but ultimately fail to successfully resolve all scenarios, and one final solution that can manage conflicts between any number of devices.

First Attempt: Store Only the Total

At first thought, it might seem that the cloud save data should simply be the number of coins in the bank. But if that data is all that's available, conflict resolution will be severely limited. The best you could do would be to pick the largest of the two numbers in case of a conflict.

Consider the scenario illustrated in Table 1. Suppose the player initially has 20 coins, and then collects 10 coins on device A and 15 coins on device B. Then device B saves the state to the cloud. When device A attempts to save, a conflict is detected. The "store only the total" conflict resolution algorithm would resolve the conflict by writing 35 (the largest of the two numbers).

Table 1. Storing only the total number of coins (failed strategy).

Event	Data on Device A	Data on Device B	Data on Cloud	Actual Total
Starting conditions	20	20	20	20
Player collects 10 coins on device A	30	20	20	30
Player collects 15 coins on device B	30	35	20	45
Device B saves state to cloud	30	35	35	45
Device A tries to save state to cloud. Conflict detected.	30	35	35	45
Device A resolves conflict by picking largest of the two numbers.	35	35	35	45

This strategy would fail—the player's bank has gone from 20 to 35, when the user actually collected a total of 25 coins (10 on device A and 15 on device B). So 10 coins were lost. Storing only the total number of coins in the cloud save is not enough to implement a robust conflict resolution algorithm.

Second Attempt: Store the Total and the Delta

A different approach is to include an additional field in the save data: the number of coins added (the delta) since the last commit. In this approach the save data can be represented by a tuple *(T,d)* where *T* is the total

number of coins and *d* is the number of coins that you just added.

With this structure, your conflict resolution algorithm has room to be more robust, as illustrated below. But this approach still doesn't give your app a reliable picture of the player's overall state.

Here is the conflict resolution algorithm for including the delta:

- **Local data:** (T, d)
- **Cloud data:** (T', d')
- **Resolved data:** (T' + d, d)

For example, when you get a conflict between the local state *(T,d)* and the cloud state *(T',d')*, you can resolve it as *(T'+d, d)*. What this means is that you are taking the delta from your local data and incorporating it into the cloud data, hoping that this will correctly account for any gold coins that were collected on the other device.

This approach might sound promising, but it breaks down in a dynamic mobile environment:

- Users might save state when the device is offline. These changes will be queued up for submission when the device comes back online.
- The way that sync works is that the most recent change overwrites any previous changes. In other words, the second write is the only one that gets sent to the cloud (this happens when the device eventually comes online), and the delta in the first write is ignored.

To illustrate, consider the scenario illustrated by Table 2. After the series of operations shown in the table, the cloud state will be (130, +5). This means the resolved state would be (140, +10). This is incorrect because in total, the user has collected 110 coins on device A and 120 coins on device B. The total should be 250 coins.

Table 2. Failure case for total+delta strategy.

Event	Data on Device A	Data on Device B	Data on Cloud	Actual Total
Starting conditions	(20, x)	(20, x)	(20, x)	20

Player collects 100 coins on device A	(120, +100)	(20, x)	(20, x)	120
Player collects 10 more coins on device A	(130, +10)	(20, x)	(20, x)	130
Player collects 115 coins on device B	(130, +10)	(125, +115)	(20, x)	245
Player collects 5 more coins on device B	(130, +10)	(130, +5)	(20, x)	250
Device B uploads its data to the cloud	(130, +10)	(130, +5)	(130, +5)	250
Device A tries to upload its data to the cloud. Conflict detected.	(130, +10)	(130, +5)	(130, +5)	250
Device A resolves the conflict by applying the local delta to the cloud total.	(140, +10)	(130, +5)	(140, +10)	250

(*): x represents data that is irrelevant to our scenario.

You might try to fix the problem by not resetting the delta after each save, so that the second save on each device accounts for all the coins collected thus far. With that change the second save made by device A would be *(130, +110)* instead of *(130, +10)*. However, you would then run into the problem illustrated in Table 3.

Table 3. Failure case for the modified algorithm.

Event	Data on Device A	Data on Device B	Data on Cloud	Actual Total
Starting conditions	(20, x)	(20, x)	(20, x)	20
Player collects 100 coins on device A	(120, +100)	(20, x)	(20, x)	120
Device A saves state to cloud	(120, +100)	(20, x)	(120, +100)	120
Player collects 10 more coins				

on device A	(130, +110)	(20, x)	(120, +100)	130
Player collects 1 coin on device B	(130, +110)	(21, +1)	(120, +100)	131
Device B attempts to save state to cloud. Conflict detected.	(130, +110)	(21, +1)	(120, +100)	131
Device B solves conflict by applying local delta to cloud total.	(130, +110)	(121, +1)	(121, +1)	131
Device A tries to upload its data to the cloud. Conflict detected.	(130, +110)	(121, +1)	(121, +1)	131
Device A resolves the conflict by applying the local delta to the cloud total.	(231, +110)	(121, +1)	(231, +110)	131

(*): x represents data that is irrelevant to our scenario.

Now you have the opposite problem: you are giving the player too many coins. The player has gained 211 coins, when in fact she has collected only 111 coins.

Solution: Store the Sub-totals per Device

Analyzing the previous attempts, it seems that what those strategies fundamentally miss is the ability to know which coins have already been counted and which coins have not been counted yet, especially in the presence of multiple consecutive commits coming from different devices.

The solution to the problem is to change the structure of your cloud save to be a dictionary that maps strings to integers. Each key-value pair in this dictionary represents a "drawer" that contains coins, and the total number of coins in the save is the sum of the values of all entries. The fundamental principle of this design is that each device has its own drawer, and only the device itself can put coins into that drawer.

The structure of the dictionary is *(A:a, B:b, C:c, ...)*, where *a* is the total

number of coins in the drawer A, *b* is the total number of coins in drawer B, and so on.

The new conflict resolution algorithm for the "drawer" solution is as follows:

- **Local data:** (A:a, B:b, C:c, ...)
- **Cloud data:** (A:a', B:b', C:c', ...)
- **Resolved data:** (A:*max*(a,a'), B:*max*(b,b'), C:*max*(c,c'), ...)

For example, if the local data is *(A:20, B:4, C:7)* and the cloud data is *(B:10, C:2, D:14)*, then the resolved data will be *(A:20, B:10, C:7, D:14)*. Note that how you apply conflict resolution logic to this dictionary data may vary depending on your app. For example, for some apps you might want to take the lower value.

To test this new algorithm, apply it to any of the test scenarios mentioned above. You will see that it arrives at the correct result.

Table 4 illustrates this, based on the scenario from Table 3. Note the following:

- In the initial state, the player has 20 coins. This is accurately reflected on each device and the cloud. This value is represented as a dictionary (X:20), where the value of X isn't significant—we don't care where this initial data came from.
- When the player collects 100 coins on device A, this change is packaged as a dictionary and saved to the cloud. The dictionary's value is 100 because that is the number of coins that the player collected on device A. There is no calculation being performed on the data at this point—device A is simply reporting the number of coins the player collected on it.
- Each new submission of coins is packaged as a dictionary associated with the device that saved it to the cloud. When the player collects 10 more coins on device A, for example, the device A dictionary value is updated to be 110.
- The net result is that the app knows the total number of coins the player has collected on each device. Thus it can easily calculate the total.

Table 4. Successful application of the key-value pair strategy.

Event	Data on Device A	Data on Device B	Data on Cloud	Actual Total
Starting conditions	(X:20, x)	(X:20, x)	(X:20, x)	20
Player collects 100 coins on device A	(X:20, A:100)	(X:20)	(X:20)	120
Device A saves state to cloud	(X:20, A:100)	(X:20)	(X:20, A:100)	120
Player collects 10 more coins on device A	(X:20, A:110)	(X:20)	(X:20, A:100)	130
Player collects 1 coin on device B	(X:20, A:110)	(X:20, B:1)	(X:20, A:100)	131
Device B attempts to save state to cloud. Conflict detected.	(X:20, A:110)	(X:20, B:1)	(X:20, A:100)	131
Device B solves conflict	(X:20, A:110)	(X:20, A:100, B:1)	(X:20, A:100, B:1)	131
Device A tries to				

upload its data to the cloud. Conflict detected.	(X:20, A:110)	(X:20, A:100, B:1)	(X:20, A:100, B:1)	131
Device A resolves the conflict	(X:20, A:110, B:1)	(X:20, A:100, B:1)	(X:20, A:110, B:1) *total 131*	131

Clean Up Your Data

There is a limit to the size of cloud save data, so in following the strategy outlined in this article, take care not to create arbitrarily large dictionaries. At first glance it may seem that the dictionary will have only one entry per device, and even the very enthusiastic user is unlikely to have thousands of them. However, obtaining a device ID is difficult and considered a bad practice, so instead you should use an installation ID, which is easier to obtain and more reliable. This means that the dictionary might have one entry for each time the user installed the application on each device. Assuming each key-value pair takes 32 bytes, and since an individual cloud save buffer can be up to 128K in size, you are safe if you have up to 4,096 entries.

In real-life situations, your data will probably be more complex than a number of coins. In this case, the number of entries in this dictionary may be much more limited. Depending on your implementation, it might make sense to store the timestamp for when each entry in the dictionary was modified. When you detect that a given entry has not been modified in the last several weeks or months, it is probably safe to transfer the coins into another entry and delete the old entry.

Next Get started Next class

Transferring Data Using Sync Adapters

Previous Next Get started

Dependencies and prerequisites

- Android 2.1 (API Level 7) or higher

You should also read

- Bound Services
- Content Providers
- Creating a Custom Account Type

Try it out

Download the sample

BasicSyncAdapter.zip

Synchronizing data between an Android device and web servers can make your application significantly more useful and compelling for your users. For example, transferring data to a web server makes a useful backup, and transferring data from a server makes it available to the user even when the device is offline. In some cases, users may find it easier to enter and edit their data in a web interface and then have that data available on their device, or they may want to collect data over time and then upload it to a central storage area.

Although you can design your own system for doing data transfers in your app, you should consider using Android's sync adapter framework.

This framework helps manage and automate data transfers, and coordinates synchronization operations across different apps. When you use this framework, you can take advantage of several features that aren't available to data transfer schemes you design yourself:

Plug-in architecture
: Allows you to add data transfer code to the system in the form of callable components.

Automated execution
: Allows you to automate data transfer based on a variety of criteria, including data changes, elapsed time, or time of day. In addition, the system adds transfers that are unable to run to a queue, and runs them when possible.

Automated network checking
: The system only runs your data transfer when the device has network connectivity.

Improved battery performance
: Allows you to centralize all of your app's data transfer tasks in one place, so that they all run at the same time. Your data transfer is also scheduled in conjunction with data transfers from other apps. These factors reduce the number of times the system has to switch on the network, which reduces battery usage.

Account management and authentication
: If your app requires user credentials or server login, you can optionally integrate account management and authentication into your data transfer.

This class shows you how to create a sync adapter and the bound `Service` that wraps it, how to provide the other components that help you plug the sync adapter into the framework, and how to run the sync adapter to run in various ways.

Note: Sync adapters run asynchronously, so you should use them with the expectation that they transfer data regularly and efficiently, but not instantaneously. If you need to do real-time data transfer, you should do it in an `AsyncTask` or an `IntentService`.

Lessons

Creating a Stub Authenticator

Learn how to add an account-handling component that the sync adapter framework expects to be part of your app. This lesson shows you how to create a stub authentication component for simplicity.

Creating a Stub Content Provider

Learn how to add a content provider component that the sync adapter framework expects to be part of your app. This lesson assumes that your app doesn't use a content provider, so it shows you how to add a stub component. If you have a content provider already in your app, you can skip this lesson.

Creating a Sync Adapter

Learn how to encapsulate your data transfer code in a component that the sync adapter framework can run automatically.

Running a Sync Adapter

Learn how to trigger and schedule data transfers using the sync adapter framework.

Next Get started Next class

Creating a Stub Authenticator

Previous Next Get started

This lesson teaches you to

1. Add a Stub Authenticator Component
2. Bind the Authenticator to the Framework
3. Add the Authenticator Metadata File
4. Declare the Authenticator in the Manifest

You should also read

- Bound Services

Try it out

Download the sample

BasicSyncAdapter.zip

The sync adapter framework assumes that your sync adapter transfers data between device storage associated with an account and server storage that requires login access. For this reason, the framework expects you to provide a component called an authenticator as part of your sync adapter. This component plugs into the Android accounts and authentication framework and provides a standard interface for handling user credentials such as login information.

Even if your app doesn't use accounts, you still need to provide an authenticator component. If you don't use accounts or server login, the information handled by the authenticator is ignored, so you can provide an authenticator component that contains stub method implementations.

You also need to provide a bound `Service` that allows the sync adapter framework to call the authenticator's methods.

This lesson shows you how to define all the parts of a stub authenticator that you need to satisfy the requirements of the sync adapter framework. If you need to provide a real authenticator that handles user accounts, read the reference documentation for `AbstractAccountAuthenticator`.

Add a Stub Authenticator Component

To add a stub authenticator component to your app, create a class that extends `AbstractAccountAuthenticator`, and then stub out the required methods, either by returning `null` or by throwing an exception.

The following snippet shows an example of a stub authenticator class:

```
/*
 * Implement AbstractAccountAuthenticator and stub out all
 * of its methods
 */
public class Authenticator extends AbstractAccountAuthenticator {
    // Simple constructor
    public Authenticator(Context context) {
        super(context);
    }
    // Editing properties is not supported
    @Override
    public Bundle editProperties(
            AccountAuthenticatorResponse r, String s) {
        throw new UnsupportedOperationException();
    }
    // Don't add additional accounts
    @Override
    public Bundle addAccount(
            AccountAuthenticatorResponse r,
            String s,
            String s2,
            String[] strings,
            Bundle bundle) throws NetworkErrorException {
        return null;
    }
    // Ignore attempts to confirm credentials
    @Override
    public Bundle confirmCredentials(
            AccountAuthenticatorResponse r,
            Account account,
            Bundle bundle) throws NetworkErrorException {
        return null;
    }
    // Getting an authentication token is not supported
    @Override
```

```
    public Bundle getAuthToken(
            AccountAuthenticatorResponse r,
            Account account,
            String s,
            Bundle bundle) throws NetworkErrorException {
        throw new UnsupportedOperationException();
    }
    // Getting a label for the auth token is not supported
    @Override
    public String getAuthTokenLabel(String s) {
        throw new UnsupportedOperationException();
    }
    // Updating user credentials is not supported
    @Override
    public Bundle updateCredentials(
            AccountAuthenticatorResponse r,
            Account account,
            String s, Bundle bundle) throws NetworkErrorException {
        throw new UnsupportedOperationException();
    }
    // Checking features for the account is not supported
    @Override
    public Bundle hasFeatures(
        AccountAuthenticatorResponse r,
        Account account, String[] strings) throws
NetworkErrorException {
        throw new UnsupportedOperationException();
    }
}
```

Bind the Authenticator to the Framework

In order for the sync adapter framework to access your authenticator, you must create a bound Service for it. This service provides an Android binder object that allows the framework to call your authenticator and pass data between the authenticator and the framework.

Since the framework starts this `Service` the first time it needs to access the authenticator, you can also use the service to instantiate the authenticator, by calling the authenticator constructor in the `Service.onCreate()` method of the service.

The following snippet shows you how to define the bound `Service`:

```
/**
 * A bound Service that instantiates the authenticator
 * when started.
 */
public class AuthenticatorService extends Service {
    ...
    // Instance field that stores the authenticator object
    private Authenticator mAuthenticator;
```

```
    @Override
    public void onCreate() {
        // Create a new authenticator object
        mAuthenticator = new Authenticator(this);
    }
    /*
     * When the system binds to this Service to make the RPC call
     * return the authenticator's IBinder.
     */
    @Override
    public IBinder onBind(Intent intent) {
        return mAuthenticator.getIBinder();
    }
}
```

Add the Authenticator Metadata File

To plug your authenticator component into the sync adapter and account frameworks, you need to provide these framework with metadata that describes the component. This metadata declares the account type you've created for your sync adapter and declares user interface elements that the system displays if you want to make your account type visible to the user. Declare this metadata in a XML file stored in the `/res/xml/` directory in your app project. You can give any name to the file, although it's usually called `authenticator.xml`.

This XML file contains a single element `<account-authenticator>` that has the following attributes:

`android:accountType`

The sync adapter framework requires each sync adapter to have an account type, in the form of a domain name. The framework uses the account type as part of the sync adapter's internal identification. For servers that require login, the account type along with a user account is sent to the server as part of the login credentials.

If your server doesn't require login, you still have to provide an account type. For the value, use a domain name that you control. While the framework uses it to manage your sync adapter, the value is not sent to your server.

`android:icon`

Pointer to a Drawable resource containing an icon. If you make the

sync adapter visible by specifying the attribute `android:userVisible="true"` in `res/xml/syncadapter.xml`, then you must provide this icon resource. It appears in the **Accounts** section of the system's Settings app.

`android:smallIcon`
: Pointer to a Drawable resource containing a small version of the icon. This resource may be used instead of `android:icon` in the **Accounts** section of the system's Settings app, depending on the screen size.

`android:label`
: Localizable string that identifies the account type to users. If you make the sync adapter visible by specifying the attribute `android:userVisible="true"` in `res/xml/syncadapter.xml`, then you should provide this string. It appears in the **Accounts** section of the system's Settings app, next to the icon you define for the authenticator.

The following snippet shows the XML file for the authenticator you created previously:

```
<?xml version="1.0" encoding="utf-8"?>
<account-authenticator
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:accountType="example.com"
        android:icon="@drawable/ic_launcher"
        android:smallIcon="@drawable/ic_launcher"
        android:label="@string/app_name"/>
```

Declare the Authenticator in the Manifest

In a previous step, you created a bound `Service` that links the authenticator to the sync adapter framework. To identify this service to the system, declare it in your app manifest by adding the following `<service>` element as a child element of `<application>`:

```
    <service

android:name="com.example.android.syncadapter.AuthenticatorService">
        <intent-filter>
            <action
android:name="android.accounts.AccountAuthenticator"/>
        </intent-filter>
        <meta-data
```

```
            android:name="android.accounts.AccountAuthenticator"
            android:resource="@xml/authenticator" />
    </service>
```

The `<intent-filter>` element sets up a filter that's triggered by the intent action `android.accounts.AccountAuthenticator`, which sent by the system to run the authenticator. When the filter is triggered, the system starts `AuthenticatorService`, the bound `Service` you have provided to wrap the authenticator.

The `<meta-data>` element declares the metadata for the authenticator. The `android:name` attribute links the meta-data to the authentication framework. The `android:resource` element specifies the name of the authenticator metadata file you created previously.

Besides an authenticator, a sync adapter also requires a content provider. If your app doesn't use a content provider already, go to the next lesson to learn how to create a stub content provider; otherwise, go to the lesson Creating a Sync Adapter.

Next Get started Next class

Creating a Stub Content Provider

Previous Next Get started

This lesson teaches you to

1. Add a Stub Content Provider
2. Declare the Provider in the Manifest

You should also read

- Content Provider Basics

Try it out

Download the sample

BasicSyncAdapter.zip

The sync adapter framework is designed to work with device data managed by the flexible and highly secure content provider framework. For this reason, the sync adapter framework expects that an app that uses the framework has already defined a content provider for its local data. If the sync adapter framework tries to run your sync adapter, and your app doesn't have a content provider, your sync adapter crashes.

If you're developing a new app that transfers data from a server to the device, you should strongly consider storing the local data in a content provider. Besides their importance for sync adapters, content providers offer a variety of security benefits and are specifically designed to handle data storage on Android systems. To learn more about creating a content provider, see Creating a Content Provider.

However, if you're already storing local data in another form, you can still use a sync adapter to handle data transfer. To satisfy the sync adapter framework requirement for a content provider, add a stub content provider to your app. A stub provider implements the content provider class, but all of its required methods return `null` or `0`. If you add a stub provider, you can then use a sync adapter to transfer data from any storage mechanism you choose.

If you already have a content provider in your app, you don't need a stub content provider. In that case, you can skip this lesson and proceed to the lesson Creating a Sync Adapter. If you don't yet have a content provider, this lesson shows you how to add a stub content provider that allows you to plug your sync adapter into the framework.

Add a Stub Content Provider

To create a stub content provider for your app, extend the class `ContentProvider` and stub out its required methods. The following snippet shows you how to create the stub provider:

```
/*
 * Define an implementation of ContentProvider that stubs out
 * all methods
 */
public class StubProvider extends ContentProvider {
    /*
     * Always return true, indicating that the
     * provider loaded correctly.
     */
    @Override
    public boolean onCreate() {
        return true;
    }
    /*
     * Return no type for MIME type
     */
    @Override
    public String getType(Uri uri) {
        return null;
    }
    /*
     * query() always returns no results
     *
     */
    @Override
    public Cursor query(
            Uri uri,
            String[] projection,
```

```
            String selection,
            String[] selectionArgs,
            String sortOrder) {
        return null;
    }
    /*
     * insert() always returns null (no URI)
     */
    @Override
    public Uri insert(Uri uri, ContentValues values) {
        return null;
    }
    /*
     * delete() always returns "no rows affected" (0)
     */
    @Override
    public int delete(Uri uri, String selection, String[]
selectionArgs) {
        return 0;
    }
    /*
     * update() always returns "no rows affected" (0)
     */
    public int update(
            Uri uri,
            ContentValues values,
            String selection,
            String[] selectionArgs) {
        return 0;
    }
}
```

Declare the Provider in the Manifest

The sync adapter framework verifies that your app has a content provider by checking that your app has declared a provider in its app manifest. To declare the stub provider in the manifest, add a `<provider>` element with the following attributes:

`android:name="com.example.android.datasync.provider.StubProvider"`
: Specifies the fully-qualified name of the class that implements the stub content provider.

`android:authorities="com.example.android.datasync.provider"`
: A URI authority that identifies the stub content provider. Make this value your app's package name with the string ".provider" appended to it. Even though you're declaring your stub provider to the system, nothing tries to access the provider itself.

`android:exported="false"`
: Determines whether other apps can access the content provider. For

your stub content provider, set the value to `false`, since there's no need to allow other apps to see the provider. This value doesn't affect the interaction between the sync adapter framework and the content provider.

`android:syncable="true"`
: Sets a flag that indicates that the provider is syncable. If you set this flag to `true`, you don't have to call `setIsSyncable()` in your code. The flag allows the sync adapter framework to make data transfers with the content provider, but transfers only occur if you do them explicitly.

The following snippet shows you how to add the `<provider>` element to the app manifest:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.android.network.sync.BasicSyncAdapter"
    android:versionCode="1"
    android:versionName="1.0" >
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
    ...
    <provider

android:name="com.example.android.datasync.provider.StubProvider"
        android:authorities="com.example.android.datasync.provider"
        android:exported="false"
        android:syncable="true"/>
    ...
    </application>
</manifest>
```

Now that you have created the dependencies required by the sync adapter framework, you can create the component that encapsulates your data transfer code. This component is called a sync adapter. The next lesson shows you how to add this component to your app.

Next Get started Next class

Creating a Sync Adapter

Previous Next Get started

This lesson teaches you to

1. Create the Sync Adapter Class
2. Bind the Sync Adapter to the Framework
3. Add the Account Required by the Framework
4. Add the Sync Adapter Metadata File
5. Declare the Sync Adapter in the Manifest

You should also read

- Bound Services
- Content Providers
- Creating a Custom Account Type

Try it out

Download the sample

BasicSyncAdapter.zip

The sync adapter component in your app encapsulates the code for the tasks that transfer data between the device and a server. Based on the scheduling and triggers you provide in your app, the sync adapter framework runs the code in the sync adapter component. To add a sync adapter component to your app, you need to add the following pieces:

Sync adapter class.
: A class that wraps your data transfer code in an interface compatible with the sync adapter framework.

Bound `Service`.
: A component that allows the sync adapter framework to run the code in your sync adapter class.

Sync adapter XML metadata file.
: A file containing information about your sync adapter. The framework reads this file to find out how to load and schedule your data transfer.

Declarations in the app manifest.
: XML that declares the bound service and points to sync adapter-specific metadata.

This lesson shows you how to define these elements.

Create a Sync Adapter Class

In this part of the lesson you learn how to create the sync adapter class that encapsulates the data transfer code. Creating the class includes extending the sync adapter base class, defining constructors for the class, and implementing the method where you define the data transfer tasks.

Extend the base sync adapter class AbstractThreadedSyncAdapter

To create the sync adapter component, start by extending `AbstractThreadedSyncAdapter` and writing its constructors. Use the constructors to run setup tasks each time your sync adapter component is created from scratch, just as you use `Activity.onCreate()` to set up an activity. For example, if your app uses a content provider to store data, use the constructors to get a `ContentResolver` instance. Since a second form of the constructor was added in Android platform version 3.0 to support the `parallelSyncs` argument, you need to create two forms of the constructor to maintain compatibility.

Note: The sync adapter framework is designed to work with sync adapter components that are singleton instances. Instantiating the sync adapter component is covered in more detail in the section Bind the Sync Adapter to the Framework.

The following example shows you how to implement AbstractThreadedSyncAdapterand its constructors:

```
/**
 * Handle the transfer of data between a server and an
 * app, using the Android sync adapter framework.
 */
public class SyncAdapter extends AbstractThreadedSyncAdapter {
    ...
    // Global variables
    // Define a variable to contain a content resolver instance
    ContentResolver mContentResolver;
    /**
     * Set up the sync adapter
     */
    public SyncAdapter(Context context, boolean autoInitialize) {
        super(context, autoInitialize);
        /*
         * If your app uses a content resolver, get an instance of
it
         * from the incoming Context
         */
        mContentResolver = context.getContentResolver();
    }
    ...
    /**
     * Set up the sync adapter. This form of the
     * constructor maintains compatibility with Android 3.0
     * and later platform versions
     */
    public SyncAdapter(
            Context context,
            boolean autoInitialize,
            boolean allowParallelSyncs) {
        super(context, autoInitialize, allowParallelSyncs);
        /*
         * If your app uses a content resolver, get an instance of
it
         * from the incoming Context
         */
        mContentResolver = context.getContentResolver();
        ...
    }
```

Add the data transfer code to onPerformSync()

The sync adapter component does not automatically do data transfer. Instead, it encapsulates your data transfer code, so that the sync adapter framework can run the data transfer in the background, without involvement from your app. When the framework is ready to sync your application's data, it invokes your implementation of the method onPerformSync().

To facilitate the transfer of data from your main app code to the sync adapter component, the sync adapter framework calls `onPerformSync()` with the following arguments:

Account
: An `Account` object associated with the event that triggered the sync adapter. If your server doesn't use accounts, you don't need to use the information in this object.

Extras
: A `Bundle` containing flags sent by the event that triggered the sync adapter.

Authority
: The authority of a content provider in the system. Your app has to have access to this provider. Usually, the authority corresponds to a content provider in your own app.

Content provider client
: A `ContentProviderClient` for the content provider pointed to by the authority argument. A `ContentProviderClient` is a lightweight public interface to a content provider. It has the same basic functionality as a `ContentResolver`. If you're using a content provider to store data for your app, you can connect to the provider with this object. Otherwise, you can ignore it.

Sync result
: A `SyncResult` object that you use to send information to the sync adapter framework.

The following snippet shows the overall structure of `onPerformSync()`:

```
    /*
     * Specify the code you want to run in the sync adapter. The
entire
     * sync adapter runs in a background thread, so you don't have
to set
     * up your own background processing.
     */
    @Override
    public void onPerformSync(
            Account account,
            Bundle extras,
            String authority,
            ContentProviderClient provider,
            SyncResult syncResult) {
    /*
     * Put the data transfer code here.
```

```
     */
    ...
    }
```

While the actual implementation of `onPerformSync()` is specific to your app's data synchronization requirements and server connection protocols, there are a few general tasks your implementation should perform:

Connecting to a server
: Although you can assume that the network is available when your data transfer starts, the sync adapter framework doesn't automatically connect to a server.

Downloading and uploading data
: A sync adapter doesn't automate any data transfer tasks. If you want to download data from a server and store it in a content provider, you have to provide the code that requests the data, downloads it, and inserts it in the provider. Similarly, if you want to send data to a server, you have to read it from a file, database, or provider, and send the necessary upload request. You also have to handle network errors that occur while your data transfer is running.

Handling data conflicts or determining how current the data is
: A sync adapter doesn't automatically handle conflicts between data on the server and data on the device. Also, it doesn't automatically detect if the data on the server is newer than the data on the device, or vice versa. Instead, you have to provide your own algorithms for handling this situation.

Clean up.
: Always close connections to a server and clean up temp files and caches at the end of your data transfer.

Note: The sync adapter framework runs `onPerformSync()` on a background thread, so you don't have to set up your own background processing.

In addition to your sync-related tasks, you should try to combine your regular network-related tasks and add them to `onPerformSync()`. By concentrating all of your network tasks in this method, you conserve the battery power that's needed to start and stop the network interfaces. To learn more about making network access more efficient, see the training

class Transferring Data Without Draining the Battery, which describes several network access tasks you can include in your data transfer code.

Bind the Sync Adapter to the Framework

You now have your data transfer code encapsulated in a sync adapter component, but you have to provide the framework with access to your code. To do this, you need to create a bound `Service` that passes a special Android binder object from the sync adapter component to the framework. With this binder object, the framework can invoke the `onPerformSync()` method and pass data to it.

Instantiate your sync adapter component as a singleton in the `onCreate()` method of the service. By instantiating the component in `onCreate()`, you defer creating it until the service starts, which happens when the framework first tries to run your data transfer. You need to instantiate the component in a thread-safe manner, in case the sync adapter framework queues up multiple executions of your sync adapter in response to triggers or scheduling.

For example, the following snippet shows you how to create a class that implements the bound `Service`, instantiates your sync adapter component, and gets the Android binder object:

```
package com.example.android.syncadapter;
/**
 * Define a Service that returns an IBinder for the
 * sync adapter class, allowing the sync adapter framework to call
 * onPerformSync().
 */
public class SyncService extends Service {
    // Storage for an instance of the sync adapter
    private static SyncAdapter sSyncAdapter = null;
    // Object to use as a thread-safe lock
    private static final Object sSyncAdapterLock = new Object();
    /*
     * Instantiate the sync adapter object.
     */
    @Override
    public void onCreate() {
        /*
         * Create the sync adapter as a singleton.
         * Set the sync adapter as syncable
         * Disallow parallel syncs
         */
        synchronized (sSyncAdapterLock) {
```

```
            if (sSyncAdapter == null) {
                sSyncAdapter = new
SyncAdapter(getApplicationContext(), true);
            }
        }
    }
    /**
     * Return an object that allows the system to invoke
     * the sync adapter.
     *
     */
    @Override
    public IBinder onBind(Intent intent) {
        /*
         * Get the object that allows external processes
         * to call onPerformSync(). The object is created
         * in the base class code when the SyncAdapter
         * constructors call super()
         */
        return sSyncAdapter.getSyncAdapterBinder();
    }
}
```

Note: To see a more detailed example of a bound service for a sync adapter, see the sample app.

Add the Account Required by the Framework

The sync adapter framework requires each sync adapter to have an account type. You declared the account type value in the section Add the Authenticator Metadata File. Now you have to set up this account type in the Android system. To set up the account type, add a dummy account that uses the account type by calling `addAccountExplicitly()`.

The best place to call the method is in the `onCreate()` method of your app's opening activity. The following code snippet shows you how to do this:

```
public class MainActivity extends FragmentActivity {
    ...
    ...
    // Constants
    // The authority for the sync adapter's content provider
    public static final String AUTHORITY =
"com.example.android.datasync.provider"
    // An account type, in the form of a domain name
    public static final String ACCOUNT_TYPE = "example.com";
    // The account name
    public static final String ACCOUNT = "dummyaccount";
```

```
    // Instance fields
    Account mAccount;
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ...
        // Create the dummy account
        mAccount = CreateSyncAccount(this);
        ...
    }
    ...
    /**
     * Create a new dummy account for the sync adapter
     *
     * @param context The application context
     */
    public static Account CreateSyncAccount(Context context) {
        // Create the account type and default account
        Account newAccount = new Account(
                ACCOUNT, ACCOUNT_TYPE);
        // Get an instance of the Android account manager
        AccountManager accountManager =
                (AccountManager) context.getSystemService(
                        ACCOUNT_SERVICE);
        /*
         * Add the account and account type, no password or user
data
         * If successful, return the Account object, otherwise
report an error.
         */
        if (accountManager.addAccountExplicitly(newAccount, null,
null))) {
            /*
             * If you don't set android:syncable="true" in
             * in your <provider> element in the manifest,
             * then call context.setIsSyncable(account, AUTHORITY,
1)
             * here.
             */
        } else {
            /*
             * The account exists or some other error occurred. Log
this, report it,
             * or handle it internally.
             */
        }
    }
    ...
}
```

Add the Sync Adapter Metadata File

To plug your sync adapter component into the framework, you need to provide the framework with metadata that describes the component and provides additional flags. The metadata specifies the account type you've

created for your sync adapter, declares a content provider authority associated with your app, controls a part of the system user interface related to sync adapters, and declares other sync-related flags. Declare this metadata in a special XML file stored in the `/res/xml/` directory in your app project. You can give any name to the file, although it's usually called `syncadapter.xml`.

This XML file contains a single XML element `<sync-adapter>` that has the following attributes:

`android:contentAuthority`
: The URI authority for your content provider. If you created a stub content provider for your app in the previous lesson Creating a Stub Content Provider, use the value you specified for the attribute `android:authorities` in the `<provider>` element you added to your app manifest. This attribute is described in more detail in the section Declare the Provider in the Manifest.
If you're transferring data from a content provider to a server with your sync adapter, this value should be the same as the content URI authority you're using for that data. This value is also one of the authorities you specify in the `android:authorities` attribute of the `<provider>` element that declares your provider in your app manifest.

`android:accountType`
: The account type required by the sync adapter framework. The value must be the same as the account type value you provided when you created the authenticator metadata file, as described in the section Add the Authenticator Metadata File. It's also the value you specified for the constant `ACCOUNT_TYPE` in the code snippet in the section Add the Account Required by the Framework.

Settings attributes

`android:userVisible`
: Sets the visibility of the sync adapter's account type. By default, the account icon and label associated with the account type are visible in the **Accounts** section of the system's Settings app, so you should make your sync adapter invisible unless you have an account type or domain that's easily associated with your app.

If you make your account type invisible, you can still allow users to control your sync adapter with a user interface in one of your app's activities.

`android:supportsUploading`
: Allows you to upload data to the cloud. Set this to `false` if your app only downloads data.

`android:allowParallelSyncs`
: Allows multiple instances of your sync adapter component to run at the same time. Use this if your app supports multiple user accounts and you want to allow multiple users to transfer data in parallel. This flag has no effect if you never run multiple data transfers.

`android:isAlwaysSyncable`
: Indicates to the sync adapter framework that it can run your sync adapter at any time you've specified. If you want to programmatically control when your sync adapter can run, set this flag to `false`, and then call `requestSync()` to run the sync adapter. To learn more about running a sync adapter, see the lesson Running a Sync Adapter

The following example shows the XML for a sync adapter that uses a single dummy account and only does downloads.

```
<?xml version="1.0" encoding="utf-8"?>
<sync-adapter
        xmlns:android="http://schemas.android.com/apk/res/android"

android:contentAuthority="com.example.android.datasync.provider"
        android:accountType="com.android.example.datasync"
        android:userVisible="false"
        android:supportsUploading="false"
        android:allowParallelSyncs="false"
        android:isAlwaysSyncable="true"/>
```

Declare the Sync Adapter in the Manifest

Once you've added the sync adapter component to your app, you have to request permissions related to using the component, and you have to declare the bound `Service` you've added.

Since the sync adapter component runs code that transfers data between

the network and the device, you need to request permission to access the Internet. In addition, your app needs to request permission to read and write sync adapter settings, so you can control the sync adapter programmatically from other components in your app. You also need to request a special permission that allows your app to use the authenticator component you created in the lesson Creating a Stub Authenticator.

To request these permissions, add the following to your app manifest as child elements of `<manifest>`:

`android.permission.INTERNET`
: Allows the sync adapter code to access the Internet so that it can download or upload data from the device to a server. You don't need to add this permission again if you were requesting it previously.

`android.permission.READ_SYNC_SETTINGS`
: Allows your app to read the current sync adapter settings. For example, you need this permission in order to call `getIsSyncable()`.

`android.permission.WRITE_SYNC_SETTINGS`
: Allows your app to control sync adapter settings. You need this permission in order to set periodic sync adapter runs using `addPeriodicSync()`. This permission is **not** required to call `requestSync()`. To learn more about running the sync adapter, see Running A Sync Adapter.

The following snippet shows how to add the permissions:

```
<manifest>
...
    <uses-permission
            android:name="android.permission.INTERNET"/>
    <uses-permission
            android:name="android.permission.READ_SYNC_SETTINGS"/>
    <uses-permission
            android:name="android.permission.WRITE_SYNC_SETTINGS"/>
    <uses-permission
            android:name="android.permission.AUTHENTICATE_ACCOUNTS"/>
...
</manifest>
```

Finally, to declare the bound `Service` that the framework uses to interact with your sync adapter, add the following XML to your app manifest as a

child element of `<application>`:

```
        <service
android:name="com.example.android.datasync.SyncService"
                android:exported="true"
                android:process=":sync">
            <intent-filter>
                <action android:name="android.content.SyncAdapter"/>
            </intent-filter>
            <meta-data android:name="android.content.SyncAdapter"
                    android:resource="@xml/syncadapter" />
        </service>
```

The `<intent-filter>` element sets up a filter that's triggered by the intent action `android.content.SyncAdapter`, sent by the system to run the sync adapter. When the filter is triggered, the system starts the bound service you've created, which in this example is `SyncService`. The attribute `android:exported="true"` allows processes other than your app (including the system) to access the `Service`. The attribute `android:process=":sync"` tells the system to run the `Service` in a global shared process named `sync`. If you have multiple sync adapters in your app they can share this process, which reduces overhead.

The `<meta-data>` element provides provides the name of the sync adapter metadata XML file you created previously. The `android:name` attribute indicates that this metadata is for the sync adapter framework. The `android:resource` element specifies the name of the metadata file.

You now have all of the components for your sync adapter. The next lesson shows you how to tell the sync adapter framework to run your sync adapter, either in response to an event or on a regular schedule.

Next Get started Next class

Running a Sync Adapter

Previous Next Get started

This lesson teaches you how to:

1. Run the Sync Adapter When Server Data Changes
2. Run the Sync Adapter When Content Provider Data Changes
3. Run the Sync Adapter After a Network Message
4. Run the Sync Adapter Periodically
5. Run the Sync Adapter On Demand

You should also read

- Content Providers

Try it out

Download the sample

BasicSyncAdapter.zip

In the previous lessons in this class, you learned how to create a sync adapter component that encapsulates data transfer code, and how to add the additional components that allow you to plug the sync adapter into the system. You now have everything you need to install an app that includes a sync adapter, but none of the code you've seen actually runs the sync adapter.

You should try to run your sync adapter based on a schedule or as the indirect result of some event. For example, you may want your sync adapter to run on a regular schedule, either after a certain period of time or at a particular time of the day. You may also want to run your sync

adapter when there are changes to data stored on the device. You should avoid running your sync adapter as the direct result of a user action, because by doing this you don't get the full benefit of the sync adapter framework's scheduling ability. For example, you should avoid providing a refresh button in your user interface.

You have the following options for running your sync adapter:

When server data changes
: Run the sync adapter in response to a message from a server, indicating that server-based data has changed. This option allows you to refresh data from the server to the device without degrading performance or wasting battery life by polling the server.

When device data changes
: Run a sync adapter when data changes on the device. This option allows you to send modified data from the device to a server, and is especially useful if you need to ensure that the server always has the latest device data. This option is straightforward to implement if you actually store data in your content provider. If you're using a stub content provider, detecting data changes may be more difficult.

When the system sends out a network message
: Run a sync adapter when the Android system sends out a network message that keeps the TCP/IP connection open; this message is a basic part of the networking framework. Using this option is one way to run the sync adapter automatically. Consider using it in conjunction with interval-based sync adapter runs.

At regular intervals
: Run a sync adapter after the expiration of an interval you choose, or run it at a certain time every day.

On demand
: Run the sync adapter in response to a user action. However, to provide the best user experience you should rely primarily on one of the more automated options. By using automated options, you conserve battery and network resources.

The rest of this lesson describes each of the options in more detail.

Run the Sync Adapter When Server Data Changes

If your app transfers data from a server and the server data changes frequently, you can use a sync adapter to do downloads in response to data changes. To run the sync adapter, have the server send a special message to a `BroadcastReceiver` in your app. In response to this message, call `ContentResolver.requestSync()` to signal the sync adapter framework to run your sync adapter.

Google Cloud Messaging (GCM) provides both the server and device components you need to make this messaging system work. Using GCM to trigger transfers is more reliable and more efficient than polling servers for status. While polling requires a `Service` that is always active, GCM uses a `BroadcastReceiver` that's activated when a message arrives. While polling at regular intervals uses battery power even if no updates are available, GCM only sends messages when needed.

Note: If you use GCM to trigger your sync adapter via a broadcast to all devices where your app is installed, remember that they receive your message at roughly the same time. This situation can cause multiple instance of your sync adapter to run at the same time, causing server and network overload. To avoid this situation for a broadcast to all devices, you should consider deferring the start of the sync adapter for a period that's unique for each device.

The following code snippet shows you how to run `requestSync()` in response to an incoming GCM message:

```
public class GcmBroadcastReceiver extends BroadcastReceiver {
    ...
    // Constants
    // Content provider authority
    public static final String AUTHORITY =
"com.example.android.datasync.provider"
    // Account type
    public static final String ACCOUNT_TYPE =
"com.example.android.datasync";
    // Account
    public static final String ACCOUNT = "default_account";
    // Incoming Intent key for extended data
    public static final String KEY_SYNC_REQUEST =
```

```
            "com.example.android.datasync.KEY_SYNC_REQUEST";
    ...
    @Override
    public void onReceive(Context context, Intent intent) {
        // Get a GCM object instance
        GoogleCloudMessaging gcm =
                GoogleCloudMessaging.getInstance(context);
        // Get the type of GCM message
        String messageType = gcm.getMessageType(intent);
        /*
         * Test the message type and examine the message contents.
         * Since GCM is a general-purpose messaging system, you
         * may receive normal messages that don't require a sync
         * adapter run.
         * The following code tests for a a boolean flag indicating
         * that the message is requesting a transfer from the
device.
         */
        if
(GoogleCloudMessaging.MESSAGE_TYPE_MESSAGE.equals(messageType)
            &&
            intent.getBooleanExtra(KEY_SYNC_REQUEST)) {
            /*
             * Signal the framework to run your sync adapter. Assume
that
             * app initialization has already created the account.
             */
            ContentResolver.requestSync(ACCOUNT, AUTHORITY, null);
            ...
        }
        ...
    }
    ...
}
```

Run the Sync Adapter When Content Provider Data Changes

If your app collects data in a content provider, and you want to update the server whenever you update the provider, you can set up your app to run your sync adapter automatically. To do this, you register an observer for the content provider. When data in your content provider changes, the content provider framework calls the observer. In the observer, call `requestSync()` to tell the framework to run your sync adapter.

Note: If you're using a stub content provider, you don't have any data in the content provider and `onChange()` is never called. In this case, you have to provide your own mechanism for detecting changes to device data. This mechanism is also responsible for calling `requestSync()` when the data changes.

To create an observer for your content provider, extend the class ContentObserver and implement both forms of its onChange() method. In onChange(), call requestSync() to start the sync adapter.

To register the observer, pass it as an argument in a call to registerContentObserver(). In this call, you also have to pass in a content URI for the data you want to watch. The content provider framework compares this watch URI to content URIs passed in as arguments to ContentResolver methods that modify your provider, such as ContentResolver.insert(). If there's a match, your implementation of ContentObserver.onChange() is called.

The following code snippet shows you how to define a ContentObserver that calls requestSync() when a table changes:

```
public class MainActivity extends FragmentActivity {
    ...
    // Constants
    // Content provider scheme
    public static final String SCHEME = "content://";
    // Content provider authority
    public static final String AUTHORITY =
"com.example.android.datasync.provider";
    // Path for the content provider table
    public static final String TABLE_PATH = "data_table";
    // Account
    public static final String ACCOUNT = "default_account";
    // Global variables
    // A content URI for the content provider's data table
    Uri mUri;
    // A content resolver for accessing the provider
    ContentResolver mResolver;
    ...
    public class TableObserver extends ContentObserver {
        /*
         * Define a method that's called when data in the
         * observed content provider changes.
         * This method signature is provided for compatibility with
         * older platforms.
         */
        @Override
        public void onChange(boolean selfChange) {
            /*
             * Invoke the method signature available as of
             * Android platform version 4.1, with a null URI.
             */
            onChange(selfChange, null);
        }
        /*
         * Define a method that's called when data in the
         * observed content provider changes.
```

```
         */
        @Override
        public void onChange(boolean selfChange, Uri changeUri) {
            /*
             * Ask the framework to run your sync adapter.
             * To maintain backward compatibility, assume that
             * changeUri is null.
            ContentResolver.requestSync(ACCOUNT, AUTHORITY, null);
        }
        ...
    }
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ...
        // Get the content resolver object for your app
        mResolver = getContentResolver();
        // Construct a URI that points to the content provider data
table
        mUri = new Uri.Builder()
                  .scheme(SCHEME)
                  .authority(AUTHORITY)
                  .path(TABLE_PATH)
                  .build();
        /*
         * Create a content observer object.
         * Its code does not mutate the provider, so set
         * selfChange to "false"
         */
        TableObserver observer = new TableObserver(false);
        /*
         * Register the observer for the data table. The table's
path
         * and any of its subpaths trigger the observer.
         */
        mResolver.registerContentObserver(mUri, true, observer);
        ...
    }
    ...
}
```

Run the Sync Adapter After a Network Message

When a network connection is available, the Android system sends out a message every few seconds to keep the device's TCP/IP connection open. This message also goes to the `ContentResolver` of each app. By calling `setSyncAutomatically()`, you can run the sync adapter whenever the `ContentResolver` receives the message.

By scheduling your sync adapter to run when the network message is

sent, you ensure that your sync adapter is always scheduled to run while the network is available. Use this option if you don't have to force a data transfer in response to data changes, but you do want to ensure your data is regularly updated. Similarly, you can use this option if you don't want a fixed schedule for your sync adapter, but you do want it to run frequently.

Since the method `setSyncAutomatically()` doesn't disable `addPeriodicSync()`, your sync adapter may be triggered repeatedly in a short period of time. If you do want to run your sync adapter periodically on a regular schedule, you should disable `setSyncAutomatically()`.

The following code snippet shows you how to configure your `ContentResolver` to run your sync adapter in response to a network message:

```
public class MainActivity extends FragmentActivity {
    ...
    // Constants
    // Content provider authority
    public static final String AUTHORITY =
"com.example.android.datasync.provider";
    // Account
    public static final String ACCOUNT = "default_account";
    // Global variables
    // A content resolver for accessing the provider
    ContentResolver mResolver;
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ...
        // Get the content resolver for your app
        mResolver = getContentResolver();
        // Turn on automatic syncing for the default account and
authority
        mResolver.setSyncAutomatically(ACCOUNT, AUTHORITY, true);
        ...
    }
    ...
}
```

Run the Sync Adapter Periodically

You can run your sync adapter periodically by setting a period of time to wait between runs, or by running it at certain times of the day, or both. Running your sync adapter periodically allows you to roughly match the

update interval of your server.

Similarly, you can upload data from the device when your server is relatively idle, by scheduling your sync adapter to run at night. Most users leave their powered on and plugged in at night, so this time is usually available. Moreover, the device is not running other tasks at the same time as your sync adapter. If you take this approach, however, you need to ensure that each device triggers a data transfer at a slightly different time. If all devices run your sync adapter at the same time, you are likely to overload your server and cell provider data networks.

In general, periodic runs make sense if your users don't need instant updates, but expect to have regular updates. Periodic runs also make sense if you want to balance the availability of up-to-date data with the efficiency of smaller sync adapter runs that don't over-use device resources.

To run your sync adapter at regular intervals, call `addPeriodicSync()`. This schedules your sync adapter to run after a certain amount of time has elapsed. Since the sync adapter framework has to account for other sync adapter executions and tries to maximize battery efficiency, the elapsed time may vary by a few seconds. Also, the framework won't run your sync adapter if the network is not available.

Notice that `addPeriodicSync()` doesn't run the sync adapter at a particular time of day. To run your sync adapter at roughly the same time every day, use a repeating alarm as a trigger. Repeating alarms are described in more detail in the reference documentation for `AlarmManager`. If you use the method `setInexactRepeating()` to set time-of-day triggers that have some variation, you should still randomize the start time to ensure that sync adapter runs from different devices are staggered.

The method `addPeriodicSync()` doesn't disable `setSyncAutomatically()`, so you may get multiple sync runs in a relatively short period of time. Also, only a few sync adapter control flags are allowed in a call to `addPeriodicSync()`; the flags that are not allowed are described in the referenced documentation for `addPeriodicSync()`.

The following code snippet shows you how to schedule periodic sync adapter runs:

```
public class MainActivity extends FragmentActivity {
    ...
    // Constants
    // Content provider authority
    public static final String AUTHORITY =
"com.example.android.datasync.provider";
    // Account
    public static final String ACCOUNT = "default_account";
    // Sync interval constants
    public static final long SECONDS_PER_MINUTE = 60L;
    public static final long SYNC_INTERVAL_IN_MINUTES = 60L;
    public static final long SYNC_INTERVAL =
            SYNC_INTERVAL_IN_MINUTES *
            SECONDS_PER_MINUTE;
    // Global variables
    // A content resolver for accessing the provider
    ContentResolver mResolver;
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ...
        // Get the content resolver for your app
        mResolver = getContentResolver();
        /*
         * Turn on periodic syncing
         */
        ContentResolver.addPeriodicSync(
                ACCOUNT,
                AUTHORITY,
                Bundle.EMPTY,
                SYNC_INTERVAL);
        ...
    }
    ...
}
```

Run the Sync Adapter On Demand

Running your sync adapter in response to a user request is the least preferable strategy for running a sync adapter. The framework is specifically designed to conserve battery power when it runs sync adapters according to a schedule. Options that run a sync in response to data changes use battery power effectively, since the power is used to provide new data.

In comparison, allowing users to run a sync on demand means that the sync runs by itself, which is inefficient use of network and power

resources. Also, providing sync on demand leads users to request a sync even if there's no evidence that the data has changed, and running a sync that doesn't refresh data is an ineffective use of battery power. In general, your app should either use other signals to trigger a sync or schedule them at regular intervals, without user input.

However, if you still want to run the sync adapter on demand, set the sync adapter flags for a manual sync adapter run, then call `ContentResolver.requestSync()`.

Run on demand transfers with the following flags:

`SYNC_EXTRAS_MANUAL`
: Forces a manual sync. The sync adapter framework ignores the existing settings, such as the flag set by `setSyncAutomatically()`.

`SYNC_EXTRAS_EXPEDITED`
: Forces the sync to start immediately. If you don't set this, the system may wait several seconds before running the sync request, because it tries to optimize battery use by scheduling many requests in a short period of time.

The following code snippet shows you how to call `requestSync()` in response to a button click:

```
public class MainActivity extends FragmentActivity {
    ...
    // Constants
    // Content provider authority
    public static final String AUTHORITY =
            "com.example.android.datasync.provider"
    // Account type
    public static final String ACCOUNT_TYPE =
"com.example.android.datasync";
    // Account
    public static final String ACCOUNT = "default_account";
    // Instance fields
    Account mAccount;
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ...
        /*
         * Create the dummy account. The code for CreateSyncAccount
         * is listed in the lesson Creating a Sync Adapter
         */
```

```
        mAccount = CreateSyncAccount(this);
        ...
    }
    /**
     * Respond to a button click by calling requestSync(). This is
an
     * asynchronous operation.
     *
     * This method is attached to the refresh button in the layout
     * XML file
     *
     * @param v The View associated with the method call,
     * in this case a Button
     */
    public void onRefreshButtonClick(View v) {
        ...
        // Pass the settings flags by inserting them in a bundle
        Bundle settingsBundle = new Bundle();
        settingsBundle.putBoolean(
                ContentResolver.SYNC_EXTRAS_MANUAL, true);
        settingsBundle.putBoolean(
                ContentResolver.SYNC_EXTRAS_EXPEDITED, true);
        /*
         * Request the sync for the default account, authority, and
         * manual sync settings
         */
        ContentResolver.requestSync(mAccount, AUTHORITY,
settingsBundle);
    }
```

Next Get started Next class

Transmitting Network Data Using Volley

Previous Next Get started

Dependencies and prerequisites

- Android 1.6 (API Level 4) or higher

Video

Volley: Easy, Fast Networking for Android

Volley is an HTTP library that makes networking for Android apps easier and most importantly, faster. Volley is available through the open AOSP repository.

Volley offers the following benefits:

- Automatic scheduling of network requests.
- Multiple concurrent network connections.
- Transparent disk and memory response caching with standard HTTP cache coherence.
- Support for request prioritization.
- Cancellation request API. You can cancel a single request, or you can set blocks or scopes of requests to cancel.
- Ease of customization, for example, for retry and backoff.
- Strong ordering that makes it easy to correctly populate your UI with data fetched asynchronously from the network.
- Debugging and tracing tools.

Volley excels at RPC-type operations used to populate a UI, such as fetching a page of search results as structured data. It integrates easily

with any protocol and comes out of the box with support for raw strings, images, and JSON. By providing built-in support for the features you need, Volley frees you from writing boilerplate code and allows you to concentrate on the logic that is specific to your app.

Volley is not suitable for large download or streaming operations, since Volley holds all responses in memory during parsing. For large download operations, consider using an alternative like `DownloadManager`.

The core Volley library is developed in the open AOSP repository at `frameworks/volley` and contains the main request dispatch pipeline as well as a set of commonly applicable utilities, available in the Volley "toolbox." The easiest way to add Volley to your project is to clone the Volley repository and set it as a library project:

1. Git clone the repository by typing the following at the command line:

   ```
   git clone
   https://android.googlesource.com/platform/frameworks/volley
   ```

2. Import the downloaded source into your app project as an Android library project (as described in Managing Projects from Eclipse with ADT, if you're using Eclipse) or make a `.jar` file.

Lessons

Sending a Simple Request
: Learn how to send a simple request using the default behaviors of Volley, and how to cancel a request.

Setting Up a RequestQueue
: Learn how to set up a `RequestQueue`, and how to implement a singleton pattern to create a `RequestQueue` that lasts the lifetime of your app.

Making a Standard Request
: Learn how to send a request using one of Volley's out-of-the-box request types (raw strings, images, and JSON).

Implementing a Custom Request
: Learn how to implement a custom request.

Next Get started Next class

Sending a Simple Request

Previous Next Get started

This lesson teaches you to

1. Add the INTERNET Permission
2. Use newRequestQueue
3. Send a Request
4. Cancel a Request

Video

Volley: Easy, Fast Networking for Android

At a high level, you use Volley by creating a `RequestQueue` and passing it `Request` objects. The `RequestQueue` manages worker threads for running the network operations, reading from and writing to the cache, and parsing responses. Requests do the parsing of raw responses and Volley takes care of dispatching the parsed response back to the main thread for delivery.

This lesson describes how to send a request using the `Volley.newRequestQueue` convenience method, which sets up a `RequestQueue` for you. See the next lesson, Setting Up a RequestQueue, for information on how to set up a `RequestQueue` yourself.

This lesson also describes how to add a request to a `RequestQueue` and cancel a request.

Add the INTERNET Permission

To use Volley, you must add the `android.permission.INTERNET` permission

to your app's manifest. Without this, your app won't be able to connect to the network.

Use newRequestQueue

Volley provides a convenience method `Volley.newRequestQueue` that sets up a `RequestQueue` for you, using default values, and starts the queue. For example:

```
final TextView mTextView = (TextView) findViewById(R.id.text);
...

// Instantiate the RequestQueue.
RequestQueue queue = Volley.newRequestQueue(this);
String url ="http://www.google.com";

// Request a string response from the provided URL.
StringRequest stringRequest = new StringRequest(Request.Method.GET,
url,
            new Response.Listener<String>() {
    @Override
    public void onResponse(String response) {
        // Display the first 500 characters of the response string.
        mTextView.setText("Response is: "+
response.substring(0,500));
    }
}, new Response.ErrorListener() {
    @Override
    public void onErrorResponse(VolleyError error) {
        mTextView.setText("That didn't work!");
    }
});
// Add the request to the RequestQueue.
queue.add(stringRequest);
```

Volley always delivers parsed responses on the main thread. Running on the main thread is convenient for populating UI controls with received data, as you can freely modify UI controls directly from your response handler, but it's especially critical to many of the important semantics provided by the library, particularly related to canceling requests.

See Setting Up a RequestQueue for a description of how to set up a `RequestQueue` yourself, instead of using the `Volley.newRequestQueue` convenience method.

Send a Request

To send a request, you simply construct one and add it to the `RequestQueue` with `add()`, as shown above. Once you add the request it moves through the pipeline, gets serviced, and has its raw response parsed and delivered.

When you call `add()`, Volley runs one cache processing thread and a pool of network dispatch threads. When you add a request to the queue, it is picked up by the cache thread and triaged: if the request can be serviced from cache, the cached response is parsed on the cache thread and the parsed response is delivered on the main thread. If the request cannot be serviced from cache, it is placed on the network queue. The first available network thread takes the request from the queue, performs the HTTP transaction, parsse the response on the worker thread, writes the response to cache, and posts the parsed response back to the main thread for delivery.

Note that expensive operations like blocking I/O and parsing/decoding are done on worker threads. You can add a request from any thread, but responses are always delivered on the main thread.

Figure 1 illustrates the life of a request:

Figure 1. Life of a request.

Cancel a Request

To cancel a request, call `cancel()` on your `Request` object. Once cancelled, Volley guarantees that your response handler will never be called. What this means in practice is that you can cancel all of your pending requests in your activity's `onStop()` method and you don't have to litter your response handlers with checks for `getActivity() == null`, whether `onSaveInstanceState()` has been called already, or other defensive boilerplate.

To take advantage of this behavior, you would typically have to track all in-flight requests in order to be able to cancel them at the appropriate

time. There is an easier way: you can associate a tag object with each request. You can then use this tag to provide a scope of requests to cancel. For example, you can tag all of your requests with the `Activity` they are being made on behalf of, and call `requestQueue.cancelAll(this)` from `onStop()`. Similarly, you could tag all thumbnail image requests in a `ViewPager` tab with their respective tabs and cancel on swipe to make sure that the new tab isn't being held up by requests from another one.

Here is an example that uses a string value for the tag:

1. Define your tag and add it to your requests.

```
public static final String TAG = "MyTag";
StringRequest stringRequest; // Assume this exists.
RequestQueue mRequestQueue;  // Assume this exists.

// Set the tag on the request.
stringRequest.setTag(TAG);

// Add the request to the RequestQueue.
mRequestQueue.add(stringRequest);
```

2. In your activity's `onStop()` method, cancel all requests that have this tag.

```
@Override
protected void onStop () {
    super.onStop();
    if (mRequestQueue != null) {
        mRequestQueue.cancelAll(TAG);
    }
}
```

Take care when canceling requests. If you are depending on your response handler to advance a state or kick off another process, you need to account for this. Again, the response handler will not be called.

Next Get started Next class

Setting Up a RequestQueue

Previous Next Get started

This lesson teaches you to

1. Set Up a Network and Cache
2. Use a Singleton Pattern

Video

Volley: Easy, Fast Networking for Android

The previous lesson showed you how to use the convenience method `Volley.newRequestQueue` to set up a `RequestQueue`, taking advantage of Volley's default behaviors. This lesson walks you through the explicit steps of creating a `RequestQueue`, to allow you to supply your own custom behavior.

This lesson also describes the recommended practice of creating a `RequestQueue` as a singleton, which makes the `RequestQueue` last the lifetime of your app.

Set Up a Network and Cache

A `RequestQueue` needs two things to do its job: a network to perform transport of the requests, and a cache to handle caching. There are standard implementations of these available in the Volley toolbox: `DiskBasedCache` provides a one-file-per-response cache with an in-memory index, and `BasicNetwork` provides a network transport based on your choice of the Apache HTTP client `android.net.http.AndroidHttpClient` or `HttpURLConnection`.

`BasicNetwork` is Volley's default network implementation. A `BasicNetwork` must be initialized with the HTTP client your app is using to connect to the network. Typically this is a `HttpURLConnection`:

- Use `android.net.http.AndroidHttpClient` for apps targeting Android API levels lower than API Level 9 (Gingerbread). Prior to Gingerbread, `HttpURLConnection` was unreliable. For more discussion of this topic, see Android's HTTP Clients.
- Use `HttpURLConnection` for apps targeting Android API Level 9 (Gingerbread) and higher.

To create an app that runs on all versions of Android, you can check the version of Android the device is running and choose the appropriate HTTP client, for example:

```
HttpStack stack;
...
// If the device is running a version >= Gingerbread...
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.GINGERBREAD) {
    // ...use HttpURLConnection for stack.
} else {
    // ...use AndroidHttpClient for stack.
}
Network network = new BasicNetwork(stack);
```

This snippet shows you the steps involved in setting up a `RequestQueue`:

```
RequestQueue mRequestQueue;

// Instantiate the cache
Cache cache = new DiskBasedCache(getCacheDir(), 1024 * 1024); // 1MB
cap

// Set up the network to use HttpURLConnection as the HTTP client.
Network network = new BasicNetwork(new HurlStack());

// Instantiate the RequestQueue with the cache and network.
mRequestQueue = new RequestQueue(cache, network);

// Start the queue
mRequestQueue.start();

String url ="http://www.myurl.com";

// Formulate the request and handle the response.
StringRequest stringRequest = new StringRequest(Request.Method.GET,
url,
        new Response.Listener<String>() {
    @Override
    public void onResponse(String response) {
        // Do something with the response
```

```
    }
},
    new Response.ErrorListener() {
        @Override
        public void onErrorResponse(VolleyError error) {
            // Handle error
    }
});

// Add the request to the RequestQueue.
mRequestQueue.add(stringRequest);
...
```

If you just need to make a one-time request and don't want to leave the thread pool around, you can create the `RequestQueue` wherever you need it and call `stop()` on the `RequestQueue` once your response or error has come back, using the `Volley.newRequestQueue()` method described in Sending a Simple Request. But the more common use case is to create the `RequestQueue` as a singleton to keep it running for the lifetime of your app, as described in the next section.

Use a Singleton Pattern

If your application makes constant use of the network, it's probably most efficient to set up a single instance of `RequestQueue` that will last the lifetime of your app. You can achieve this in various ways. The recommended approach is to implement a singleton class that encapsulates `RequestQueue` and other Volley functionality. Another approach is to subclass `Application` and set up the `RequestQueue` in `Application.onCreate()`. But this approach is discouraged; a static singleton can provide the same functionality in a more modular way.

A key concept is that the `RequestQueue` must be instantiated with the `Application` context, not an `Activity` context. This ensures that the `RequestQueue` will last for the lifetime of your app, instead of being recreated every time the activity is recreated (for example, when the user rotates the device).

Here is an example of a singleton class that provides `RequestQueue` and `ImageLoader` functionality:

```
public class MySingleton {
    private static MySingleton mInstance;
```

```
    private RequestQueue mRequestQueue;
    private ImageLoader mImageLoader;
    private static Context mCtx;

    private MySingleton(Context context) {
        mCtx = context;
        mRequestQueue = getRequestQueue();

        mImageLoader = new ImageLoader(mRequestQueue,
                new ImageLoader.ImageCache() {
            private final LruCache<String, Bitmap>
                    cache = new LruCache<String, Bitmap>(20);

            @Override
            public Bitmap getBitmap(String url) {
                return cache.get(url);
            }

            @Override
            public void putBitmap(String url, Bitmap bitmap) {
                cache.put(url, bitmap);
            }
        });
    }

    public static synchronized MySingleton getInstance(Context
context) {
        if (mInstance == null) {
            mInstance = new MySingleton(context);
        }
        return mInstance;
    }

    public RequestQueue getRequestQueue() {
        if (mRequestQueue == null) {
            // getApplicationContext() is key, it keeps you from
leaking the
            // Activity or BroadcastReceiver if someone passes one
in.
            mRequestQueue =
Volley.newRequestQueue(mCtx.getApplicationContext());
        }
        return mRequestQueue;
    }

    public <T> void addToRequestQueue(Request<T> req) {
        getRequestQueue().add(req);
    }

    public ImageLoader getImageLoader() {
        return mImageLoader;
    }
}
```

Here are some examples of performing `RequestQueue` operations using the singleton class:

```
// Get a RequestQueue
RequestQueue queue =
```

```
MySingleton.getInstance(this.getApplicationContext()).
    getRequestQueue();
...

// Add a request (in this example, called stringRequest) to your
RequestQueue.
MySingleton.getInstance(this).addToRequestQueue(stringRequest);
```

Next Get started Next class

Making a Standard Request

Previous Next Get started

This lesson teaches you to

1. Request an Image
2. Request JSON

Video

Volley: Easy, Fast Networking for Android

This lesson describes how to use the common request types that Volley supports:

- `StringRequest`. Specify a URL and receive a raw string in response. See Setting Up a Request Queue for an example.
- `ImageRequest`. Specify a URL and receive an image in response.
- `JsonObjectRequest` and `JsonArrayRequest` (both subclasses of `JsonRequest`). Specify a URL and get a JSON object or array (respectively) in response.

If your expected response is one of these types, you probably won't have to implement a custom request. This lesson describes how to use these standard request types. For information on how to implement your own custom request, see Implementing a Custom Request.

Request an Image

Volley offers the following classes for requesting images. These classes layer on top of each other to offer different levels of support for processing images:

- `ImageRequest`—a canned request for getting an image at a given URL and calling back with a decoded bitmap. It also provides convenience features like specifying a size to resize to. Its main benefit is that Volley's thread scheduling ensures that expensive image operations (decoding, resizing) automatically happen on a worker thread.
- `ImageLoader`—a helper class that handles loading and caching images from remote URLs. `ImageLoader` is a an orchestrator for large numbers of `ImageRequest`s, for example when putting multiple thumbnails in a `ListView`. `ImageLoader` provides an in-memory cache to sit in front of the normal Volley cache, which is important to prevent flickering. This makes it possible to achieve a cache hit without blocking or deferring off the main thread, which is impossible when using disk I/O. `ImageLoader` also does response coalescing, without which almost every response handler would set a bitmap on a view and cause a layout pass per image. Coalescing makes it possible to deliver multiple responses simultaneously, which improves performance.
- `NetworkImageView`—builds on `ImageLoader` and effectively replaces `ImageView` for situations where your image is being fetched over the network via URL. `NetworkImageView` also manages canceling pending requests if the view is detached from the hierarchy.

Use ImageRequest

Here is an example of using `ImageRequest`. It retrieves the image specified by the URL and displays it in the app. Note that this snippet interacts with the `RequestQueue` through a singleton class (see Setting Up a RequestQueue for more discussion of this topic):

```
ImageView mImageView;
String url = "http://i.imgur.com/7spzG.png";
mImageView = (ImageView) findViewById(R.id.myImage);
...

// Retrieves an image specified by the URL, displays it in the UI.
ImageRequest request = new ImageRequest(url,
    new Response.Listener<Bitmap>() {
        @Override
        public void onResponse(Bitmap bitmap) {
            mImageView.setImageBitmap(bitmap);
        }
```

```
    }, 0, 0, null,
    new Response.ErrorListener() {
        public void onErrorResponse(VolleyError error) {
            mImageView.setImageResource(R.drawable.image_load_error);
        }
    });
// Access the RequestQueue through your singleton class.
MySingleton.getInstance(this).addToRequestQueue(request);
```

Use ImageLoader and NetworkImageView

You can use `ImageLoader` and `NetworkImageView` in concert to efficiently manage the display of multiple images, such as in a `ListView`. In your layout XML file, you use `NetworkImageView` in much the same way you would use `ImageView`, for example:

```
<com.android.volley.toolbox.NetworkImageView
        android:id="@+id/networkImageView"
        android:layout_width="150dp"
        android:layout_height="170dp"
        android:layout_centerHorizontal="true" />
```

You can use `ImageLoader` by itself to display an image, for example:

```
ImageLoader mImageLoader;
ImageView mImageView;
// The URL for the image that is being loaded.
private static final String IMAGE_URL =
    "http://developer.android.com/images/training/system-ui.png";
...
mImageView = (ImageView) findViewById(R.id.regularImageView);

// Get the ImageLoader through your singleton class.
mImageLoader = MySingleton.getInstance(this).getImageLoader();
mImageLoader.get(IMAGE_URL, ImageLoader.getImageListener(mImageView,
         R.drawable.def_image, R.drawable.err_image));
```

However, `NetworkImageView` can do this for you if all you're doing is populating an `ImageView`. For example:

```
ImageLoader mImageLoader;
NetworkImageView mNetworkImageView;
private static final String IMAGE_URL =
    "http://developer.android.com/images/training/system-ui.png";
...

// Get the NetworkImageView that will display the image.
mNetworkImageView = (NetworkImageView)
findViewById(R.id.networkImageView);

// Get the ImageLoader through your singleton class.
mImageLoader = MySingleton.getInstance(this).getImageLoader();
```

```
// Set the URL of the image that should be loaded into this view,
and
// specify the ImageLoader that will be used to make the request.
mNetworkImageView.setImageUrl(IMAGE_URL, mImageLoader);
```

The above snippets access the `RequestQueue` and the `ImageLoader` through a singleton class, as described in Setting Up a RequestQueue. This approach ensures that your app creates single instances of these classes that last the lifetime of your app. The reason that this is important for `ImageLoader` (the helper class that handles loading and caching images) is that the main function of the in-memory cache is to allow for flickerless rotation. Using a singleton pattern allows the bitmap cache to outlive the activity. If instead you create the `ImageLoader` in an activity, the `ImageLoader` would be recreated along with the activity every time the user rotates the device. This would cause flickering.

Example LRU cache

The Volley toolbox provides a standard cache implementation via the `DiskBasedCache` class. This class caches files directly onto the hard disk in the specified directory. But to use `ImageLoader`, you should provide a custom in-memory LRU bitmap cache that implements the `ImageLoader.ImageCache` interface. You may want to set up your cache as a singleton; for more discussion of this topic, see Setting Up a RequestQueue.

Here is a sample implementation for an in-memory `LruBitmapCache` class. It extends the `LruCache` class and implements the `ImageLoader.ImageCache` interface:

```
import android.graphics.Bitmap;
import android.support.v4.util.LruCache;
import android.util.DisplayMetrics;
import com.android.volley.toolbox.ImageLoader.ImageCache;

public class LruBitmapCache extends LruCache<String, Bitmap>
        implements ImageCache {

    public LruBitmapCache(int maxSize) {
        super(maxSize);
    }

    public LruBitmapCache(Context ctx) {
        this(getCacheSize(ctx));
    }
```

```
    @Override
    protected int sizeOf(String key, Bitmap value) {
        return value.getRowBytes() * value.getHeight();
    }

    @Override
    public Bitmap getBitmap(String url) {
        return get(url);
    }

    @Override
    public void putBitmap(String url, Bitmap bitmap) {
        put(url, bitmap);
    }

    // Returns a cache size equal to approximately three screens
worth of images.
    public static int getCacheSize(Context ctx) {
        final DisplayMetrics displayMetrics = ctx.getResources().
                getDisplayMetrics();
        final int screenWidth = displayMetrics.widthPixels;
        final int screenHeight = displayMetrics.heightPixels;
        // 4 bytes per pixel
        final int screenBytes = screenWidth * screenHeight * 4;

        return screenBytes * 3;
    }
}
```

Here is an example of how to instantiate an `ImageLoader` to use this cache:

```
RequestQueue mRequestQueue; // assume this exists.
ImageLoader mImageLoader = new ImageLoader(mRequestQueue, new
LruBitmapCache(
            LruBitmapCache.getCacheSize()));
```

Request JSON

Volley provides the following classes for JSON requests:

- `JsonArrayRequest`—A request for retrieving a `JSONArray` response body at a given URL.
- `JsonObjectRequest`—A request for retrieving a `JSONObject` response body at a given URL, allowing for an optional `JSONObject` to be passed in as part of the request body.

Both classes are based on the common base class `JsonRequest`. You use them following the same basic pattern you use for other types of

requests. For example, this snippet fetches a JSON feed and displays it as text in the UI:

```
TextView mTxtDisplay;
ImageView mImageView;
mTxtDisplay = (TextView) findViewById(R.id.txtDisplay);
String url = "http://my-json-feed";

JsonObjectRequest jsObjRequest = new JsonObjectRequest
        (Request.Method.GET, url, null, new
Response.Listener<JSONObject>() {

    @Override
    public void onResponse(JSONObject response) {
        mTxtDisplay.setText("Response: " + response.toString());
    }
}, new Response.ErrorListener() {

    @Override
    public void onErrorResponse(VolleyError error) {
        // TODO Auto-generated method stub

    }
});

// Access the RequestQueue through your singleton class.
MySingleton.getInstance(this).addToRequestQueue(jsObjRequest);
```

For an example of implementing a custom JSON request based on Gson, see the next lesson, Implementing a Custom Request.

Next Get started Next class

Implementing a Custom Request

Previous Next Get started

This lesson teaches you to

1. Write a Custom Request

Video

Volley: Easy, Fast Networking for Android

This lesson describes how to implement your own custom request types, for types that don't have out-of-the-box Volley support.

Write a Custom Request

Most requests have ready-to-use implementations in the toolbox; if your response is a string, image, or JSON, you probably won't need to implement a custom `Request`.

For cases where you do need to implement a custom request, this is all you need to do:

- Extend the `Request<T>` class, where `<T>` represents the type of parsed response the request expects. So if your parsed response is a string, for example, create your custom request by extending `Request<String>`. See the Volley toolbox classes `StringRequest` and `ImageRequest` for examples of extending `Request<T>`.
- Implement the abstract methods `parseNetworkResponse()` and `deliverResponse()`, described in more detail below.

parseNetworkResponse

A `Response` encapsulates a parsed response for delivery, for a given type (such as string, image, or JSON). Here is a sample implementation of `parseNetworkResponse()`:

```
@Override
protected Response<T> parseNetworkResponse(
        NetworkResponse response) {
    try {
        String json = new String(response.data,
        HttpHeaderParser.parseCharset(response.headers));
    return Response.success(gson.fromJson(json, clazz),
    HttpHeaderParser.parseCacheHeaders(response));
    }
    // handle errors
...
}
```

Note the following:

- `parseNetworkResponse()` takes as its parameter a `NetworkResponse`, which contains the response payload as a byte[], HTTP status code, and response headers.
- Your implementation must return a `Response<T>`, which contains your typed response object and cache metadata or an error, such as in the case of a parse failure.

If your protocol has non-standard cache semantics, you can build a `Cache.Entry` yourself, but most requests are fine with something like this:

```
return Response.success(myDecodedObject,
        HttpHeaderParser.parseCacheHeaders(response));
```

Volley calls `parseNetworkResponse()` from a worker thread. This ensures that expensive parsing operations, such as decoding a JPEG into a Bitmap, don't block the UI thread.

deliverResponse

Volley calls you back on the main thread with the object you returned in `parseNetworkResponse()`. Most requests invoke a callback interface here, for example:

```
protected void deliverResponse(T response) {
        listener.onResponse(response);
```

Example: GsonRequest

Gson is a library for converting Java objects to and from JSON using reflection. You can define Java objects that have the same names as their corresponding JSON keys, pass Gson the class object, and Gson will fill in the fields for you. Here's a complete implementation of a Volley request that uses Gson for parsing:

```
public class GsonRequest<T> extends Request<T> {
    private final Gson gson = new Gson();
    private final Class<T> clazz;
    private final Map<String, String> headers;
    private final Listener<T> listener;

    /**
     * Make a GET request and return a parsed object from JSON.
     *
     * @param url URL of the request to make
     * @param clazz Relevant class object, for Gson's reflection
     * @param headers Map of request headers
     */
    public GsonRequest(String url, Class<T> clazz, Map<String,
String> headers,
            Listener<T> listener, ErrorListener errorListener) {
        super(Method.GET, url, errorListener);
        this.clazz = clazz;
        this.headers = headers;
        this.listener = listener;
    }

    @Override
    public Map<String, String> getHeaders() throws AuthFailureError
{
        return headers != null ? headers : super.getHeaders();
    }

    @Override
    protected void deliverResponse(T response) {
        listener.onResponse(response);
    }

    @Override
    protected Response<T> parseNetworkResponse(NetworkResponse
response) {
        try {
            String json = new String(
                    response.data,
                    HttpHeaderParser.parseCharset(response.headers));
            return Response.success(
                    gson.fromJson(json, clazz),
                    HttpHeaderParser.parseCacheHeaders(response));
        } catch (UnsupportedEncodingException e) {
            return Response.error(new ParseError(e));
        } catch (JsonSyntaxException e) {
            return Response.error(new ParseError(e));
        }
```

```
    }
}
```

Volley provides ready-to-use `JsonArrayRequest` and `JsonArrayObject` classes if you prefer to take that approach. See Using Standard Request Types for more information.

Next Get started Next class

Building Apps with Location & Maps

These classes teach you how to add user location and mapping information to your app. Make your app more helpful and relevant to users by providing information about where they are and the world around them.

Making Your App Location-Aware

Previous Next Get started

Dependencies and prerequisites

- Google Play services client library (latest version)
- Android version 2.2 (API level 8) or later

You should also read

- Set Up Google Play Services SDK

One of the unique features of mobile applications is location awareness. Mobile users take their devices with them everywhere, and adding location awareness to your app offers users a more contextual experience. The location APIs available in Google Play services facilitate adding location awareness to your app with automated location tracking, geofencing, and activity recognition.

The Google Play services location APIs are preferred over the Android framework location APIs (android.location) as a way of adding location awareness to your app. If you are currently using the Android framework location APIs, you are strongly encouraged to switch to the Google Play services location APIs as soon as possible.

This class shows you how to use the Google Play services location APIs in your app to get the current location, get periodic location updates, and look up addresses. The class includes sample apps and code snippets that you can use as a starting point for adding location awareness to your app.

Note: Since this class is based on the Google Play services client library, make sure you install the latest version before using the sample apps or

code snippets. To learn how to set up the client library with the latest version, see Setup in the Google Play services guide.

Lessons

Getting the Last Known Location
Learn how to retrieve the last known location of an Android device, which is usually equivalent to the user's current location.

Receiving Location Updates
Learn how to request and receive periodic location updates.

Displaying a Location Address
Learn how to convert a location's latitude and longitude into an address (reverse geocoding).

Creating and Monitoring Geofences
Learn how to define one or more geographic areas as locations of interest, called geofences, and detect when the user is close to or inside a geofence.

Next Get started Next class

Getting the Last Known Location

Previous Next Get started

This lesson teaches you how to

1. Set Up Google Play Services
2. Specify App Permissions
3. Connect to Google Play Services
4. Get the Last Known Location

You should also read

- Setting up Google Play Services

Try it out

- BasicLocationSample

Using the Google Play services location APIs, your app can request the last known location of the user's device. In most cases, you are interested in the user's current location, which is usually equivalent to the last known location of the device.

Specifically, use the fused location provider to retrieve the device's last known location. The fused location provider is one of the location APIs in Google Play services. It manages the underlying location technology and provides a simple API so that you can specify requirements at a high level, like high accuracy or low power. It also optimizes the device's use of battery power.

This lesson shows you how to make a single request for the location of a device using the `getLastLocation()` method in the fused location

provider.

Set Up Google Play Services

To access the fused location provider, your app's development project must include Google Play services. Download and install the Google Play services component via the SDK Manager and add the library to your project. For details, see the guide to Setting Up Google Play Services.

Specify App Permissions

Apps that use location services must request location permissions. Android offers two location permissions: `ACCESS_COARSE_LOCATION` and `ACCESS_FINE_LOCATION`. The permission you choose determines the accuracy of the location returned by the API. If you specify `ACCESS_COARSE_LOCATION`, the API returns a location with an accuracy approximately equivalent to a city block.

This lesson requires only coarse location. Request this permission with the `uses-permission` element in your app manifest, as shown in the following example:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.google.android.gms.location.sample.basiclocationsample"
>

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

Connect to Google Play Services

To connect to the API, you need to create an instance of the Google Play services API client. For details about using the client, see the guide to Accessing Google APIs.

In your activity's `onCreate()` method, create an instance of Google API Client using `GoogleApiClient.Builder`. Use the builder to add the

`LocationServices` API.

The sample app defines a `buildGoogleApiClient()` method, called from the activity's `onCreate()` method, which includes the following code.

```
protected synchronized void buildGoogleApiClient() {
    mGoogleApiClient = new GoogleApiClient.Builder(this)
        .addConnectionCallbacks(this)
        .addOnConnectionFailedListener(this)
        .addApi(LocationServices.API)
        .build();
}
```

Get the Last Known Location

Once you have connected to Google Play services and the location services API, you can get the last known location of a user's device. When your app is connected to these you can use the fused location provider's `getLastLocation()` method to retrieve the device location. The precision of the location returned by this call is determined by the permission setting you put in your app manifest, as described in the Specify App Permissions section of this document.

To request the last known location, call the `getLastLocation()` method, passing it your instance of the `GoogleApiClient` object. Do this in the `onConnected()` callback provided by Google API Client, which is called when the client is ready. The following code sample illustrates the request and a simple handling of the response:

```
public class MainActivity extends ActionBarActivity implements
        ConnectionCallbacks, OnConnectionFailedListener {
    ...
    @Override
    public void onConnected(Bundle connectionHint) {
        mLastLocation =
LocationServices.FusedLocationApi.getLastLocation(
                mGoogleApiClient);
        if (mLastLocation != null) {

mLatitudeText.setText(String.valueOf(mLastLocation.getLatitude()));

mLongitudeText.setText(String.valueOf(mLastLocation.getLongitude()));
        }
    }
}
```

The `getLastLocation()` method returns a `Location` object from which you

can retrieve the latitude and longitude coordinates of a geographic location. The location object returned may be null in rare cases when the location is not available.

The next lesson, Receiving Location Updates, shows you how to receive periodic location updates.

Next Get started Next class

Receiving Location Updates

Previous Next Get started

This lesson teaches you how to

1. Connect to Location Services
2. Set Up a Location Request
3. Request Location Updates
4. Define the Location Update Callback
5. Stop Location Updates
6. Save the State of the Activity

You should also read

- Setting up Google Play Services
- Getting the Last Known Location

Try it out

- LocationUpdates

If your app can continuously track location, it can deliver more relevant information to the user. For example, if your app helps the user find their way while walking or driving, or if your app tracks the location of assets, it needs to get the location of the device at regular intervals. As well as the geographical location (latitude and longitude), you may want to give the user further information such as the bearing (horizontal direction of travel), altitude, or velocity of the device. This information, and more, is available in the `Location` object that your app can retrieve from the fused location provider.

While you can get a device's location with `getLastLocation()`, as

illustrated in the lesson on Getting the Last Known Location, a more direct approach is to request periodic updates from the fused location provider. In response, the API updates your app periodically with the best available location, based on the currently-available location providers such as WiFi and GPS (Global Positioning System). The accuracy of the location is determined by the providers, the location permissions you've requested, and the options you set in the location request.

This lesson shows you how to request regular updates about a device's location using the `requestLocationUpdates()` method in the fused location provider.

Connect to Location Services

Location services for apps are provided through Google Play services and the fused location provider. In order to use these services, you connect your app using the Google API Client and then request location updates. For details on connecting with the `GoogleApiClient`, follow the instructions in Getting the Last Known Location, including requesting the current location.

The last known location of the device provides a handy base from which to start, ensuring that the app has a known location before starting the periodic location updates. The lesson on Getting the Last Known Location shows you how to get the last known location by calling `getLastLocation()`. The snippets in the following sections assume that your app has already retrieved the last known location and stored it as a `Location` object in the global variable `mCurrentLocation`.

Apps that use location services must request location permissions. In this lesson you require fine location detection, so that your app can get as precise a location as possible from the available location providers. Request this permission with the `uses-permission` element in your app manifest, as shown in the following example:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.google.android.gms.location.sample.locationupdates"
>
```

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

Set Up a Location Request

To store parameters for requests to the fused location provider, create a `LocationRequest`. The parameters determine the levels of accuracy requested. For details of all the options available in the location request, see the `LocationRequest` class reference. This lesson sets the update interval, fastest update interval, and priority, as described below:

Update interval

`setInterval()` - This method sets the rate in milliseconds at which your app prefers to receive location updates. Note that the location updates may be faster than this rate if another app is receiving updates at a faster rate, or slower than this rate, or there may be no updates at all (if the device has no connectivity, for example).

Fastest update interval

`setFastestInterval()` - This method sets the **fastest** rate in milliseconds at which your app can handle location updates. You need to set this rate because other apps also affect the rate at which updates are sent. The Google Play services location APIs send out updates at the fastest rate that any app has requested with `setInterval()`. If this rate is faster than your app can handle, you may encounter problems with UI flicker or data overflow. To prevent this, call `setFastestInterval()` to set an upper limit to the update rate.

Priority

`setPriority()` - This method sets the priority of the request, which gives the Google Play services location services a strong hint about which location sources to use. The following values are supported:

- `PRIORITY_BALANCED_POWER_ACCURACY` - Use this setting to request location precision to within a city block, which is an accuracy of approximately 100 meters. This is considered a coarse level of accuracy, and is likely to consume less power. With this setting,

the location services are likely to use WiFi and cell tower positioning. Note, however, that the choice of location provider depends on many other factors, such as which sources are available.

- PRIORITY_HIGH_ACCURACY - Use this setting to request the most precise location possible. With this setting, the location services are more likely to use GPS (Global Positioning System) to determine the location.
- PRIORITY_LOW_POWER - Use this setting to request city-level precision, which is an accuracy of approximately 10 kilometers. This is considered a coarse level of accuracy, and is likely to consume less power.
- PRIORITY_NO_POWER - Use this setting if you need negligible impact on power consumption, but want to receive location updates when available. With this setting, your app does not trigger any location updates, but receives locations triggered by other apps.

Create the location request and set the parameters as shown in this code sample:

```
protected void createLocationRequest() {
    LocationRequest mLocationRequest = new LocationRequest();
    mLocationRequest.setInterval(10000);
    mLocationRequest.setFastestInterval(5000);

mLocationRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);
}
```

The priority of PRIORITY_HIGH_ACCURACY, combined with the ACCESS_FINE_LOCATION permission setting that you've defined in the app manifest, and a fast update interval of 5000 milliseconds (5 seconds), causes the fused location provider to return location updates that are accurate to within a few feet. This approach is appropriate for mapping apps that display the location in real time.

Performance hint: If your app accesses the network or does other long-running work after receiving a location update, adjust the fastest interval to a slower value. This adjustment prevents your app from receiving updates it can't use. Once the long-running work is done, set

the fastest interval back to a fast value.

Request Location Updates

Now that you've set up a location request containing your app's requirements for the location updates, you can start the regular updates by calling `requestLocationUpdates()`. Do this in the `onConnected()` callback provided by Google API Client, which is called when the client is ready.

Depending on the form of the request, the fused location provider either invokes the `LocationListener.onLocationChanged()` callback method and passes it a `Location` object, or issues a `PendingIntent` that contains the location in its extended data. The accuracy and frequency of the updates are affected by the location permissions you've requested and the options you set in the location request object.

This lesson shows you how to get the update using the `LocationListener` callback approach. Call `requestLocationUpdates()`, passing it your instance of the `GoogleApiClient`, the `LocationRequest` object, and a `LocationListener`. Define a `startLocationUpdates()` method, called from the `onConnected()` callback, as shown in the following code sample:

```
@Override
public void onConnected(Bundle connectionHint) {
    ...
    if (mRequestingLocationUpdates) {
        startLocationUpdates();
    }
}

protected void startLocationUpdates() {
    LocationServices.FusedLocationApi.requestLocationUpdates(
            mGoogleApiClient, mLocationRequest, this);
}
```

Notice that the above code snippet refers to a boolean flag, `mRequestingLocationUpdates`, used to track whether the user has turned location updates on or off. For more about retaining the value of this flag across instances of the activity, see Save the State of the Activity.

Define the Location Update Callback

The fused location provider invokes the `LocationListener.onLocationChanged()` callback method. The incoming argument is a `Location` object containing the location's latitude and longitude. The following snippet shows how to implement the `LocationListener` interface and define the method, then get the timestamp of the location update and display the latitude, longitude and timestamp on your app's user interface:

```
public class MainActivity extends ActionBarActivity implements
        ConnectionCallbacks, OnConnectionFailedListener,
LocationListener {
    ...
    @Override
    public void onLocationChanged(Location location) {
        mCurrentLocation = location;
        mLastUpdateTime = DateFormat.getTimeInstance().format(new
Date());
        updateUI();
    }

    private void updateUI() {

mLatitudeTextView.setText(String.valueOf(mCurrentLocation.getLatitude()


mLongitudeTextView.setText(String.valueOf(mCurrentLocation.getLongitude

        mLastUpdateTimeTextView.setText(mLastUpdateTime);
    }
}
```

Stop Location Updates

Consider whether you want to stop the location updates when the activity is no longer in focus, such as when the user switches to another app or to a different activity in the same app. This can be handy to reduce power consumption, provided the app doesn't need to collect information even when it's running in the background. This section shows how you can stop the updates in the activity's `onPause()` method.

To stop location updates, call `removeLocationUpdates()`, passing it your instance of the `GoogleApiClient` object and a `LocationListener`, as shown in the following code sample:

```
@Override
protected void onPause() {
    super.onPause();
```

```
    stopLocationUpdates();
}

protected void stopLocationUpdates() {
    LocationServices.FusedLocationApi.removeLocationUpdates(
            mGoogleApiClient, this);
}
```

Use a boolean, `mRequestingLocationUpdates`, to track whether location updates are currently turned on. In the activity's `onResume()` method, check whether location updates are currently active, and activate them if not:

```
@Override
public void onResume() {
    super.onResume();
    if (mGoogleApiClient.isConnected() &&
!mRequestingLocationUpdates) {
        startLocationUpdates();
    }
}
```

Save the State of the Activity

A change to the device's configuration, such as a change in screen orientation or language, can cause the current activity to be destroyed. Your app must therefore store any information it needs to recreate the activity. One way to do this is via an instance state stored in a `Bundle` object.

The following code sample shows how to use the activity's `onSaveInstanceState()` callback to save the instance state:

```
public void onSaveInstanceState(Bundle savedInstanceState) {
    savedInstanceState.putBoolean(REQUESTING_LOCATION_UPDATES_KEY,
            mRequestingLocationUpdates);
    savedInstanceState.putParcelable(LOCATION_KEY, mCurrentLocation);
    savedInstanceState.putString(LAST_UPDATED_TIME_STRING_KEY,
mLastUpdateTime);
    super.onSaveInstanceState(savedInstanceState);
}
```

Define an `updateValuesFromBundle()` method to restore the saved values from the previous instance of the activity, if they're available. Call the method from the activity's `onCreate()` method, as shown in the following code sample:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    ...
    updateValuesFromBundle(savedInstanceState);
}

private void updateValuesFromBundle(Bundle savedInstanceState) {
    if (savedInstanceState != null) {
        // Update the value of mRequestingLocationUpdates from the
Bundle, and
        // make sure that the Start Updates and Stop Updates buttons
are
        // correctly enabled or disabled.
        if
(savedInstanceState.keySet().contains(REQUESTING_LOCATION_UPDATES_KEY))
{
            mRequestingLocationUpdates =
savedInstanceState.getBoolean(
                    REQUESTING_LOCATION_UPDATES_KEY);
            setButtonsEnabledState();
        }

        // Update the value of mCurrentLocation from the Bundle and
update the
        // UI to show the correct latitude and longitude.
        if (savedInstanceState.keySet().contains(LOCATION_KEY)) {
            // Since LOCATION_KEY was found in the Bundle, we can be
sure that
            // mCurrentLocationis not null.
            mCurrentLocation =
savedInstanceState.getParcelable(LOCATION_KEY);
        }

        // Update the value of mLastUpdateTime from the Bundle and
update the UI.
        if
(savedInstanceState.keySet().contains(LAST_UPDATED_TIME_STRING_KEY))
{
            mLastUpdateTime = savedInstanceState.getString(
                    LAST_UPDATED_TIME_STRING_KEY);
        }
        updateUI();
    }
}
```

For more about saving instance state, see the Android Activity class reference.

Note: For a more persistent storage, you can store the user's preferences in your app's `SharedPreferences`. Set the shared preference in your activity's `onPause()` method, and retrieve the preference in `onResume()`. For more information about saving preferences, read Saving Key-Value Sets.

The next lesson, Displaying a Location Address, shows you how to

display the street address for a given location.

Next Get started Next class

Displaying a Location Address

Previous Next Get started

This lesson teaches you how to

1. Get a Geographic Location
2. Define an Intent Service to Fetch the Address
3. Start the Intent Service
4. Receive the Geocoding Results

You should also read

- Setting up Google Play Services
- Getting the Last Known Location
- Receiving Location Updates

Try it out

- LocationAddress

The lessons Getting the Last Known Location and Receiving Location Updates describe how to get the user's location in the form of a `Location` object that contains latitude and longitude coordinates. Although latitude and longitude are useful for calculating distance or displaying a map position, in many cases the address of the location is more useful. For example, if you want to let your users know where they are or what is close by, a street address is more meaningful than the geographic coordinates (latitude/longitude) of the location.

Using the `Geocoder` class in the Android framework location APIs, you can convert an address to the corresponding geographic coordinates. This process is called *geocoding*. Alternatively, you can convert a geographic

location to an address. The address lookup feature is also known as *reverse geocoding*.

This lesson shows you how to use the `getFromLocation()` method to convert a geographic location to an address. The method returns an estimated street address corresponding to a given latitude and longitude.

Get a Geographic Location

The last known location of the device is a useful starting point for the address lookup feature. The lesson on Getting the Last Known Location shows you how to use the `getLastLocation()` method provided by the fused location provider to find the latest location of the device.

To access the fused location provider, you need to create an instance of the Google Play services API client. To learn how to connect your client, see Connect to Google Play Services.

In order for the fused location provider to retrieve a precise street address, set the location permission in your app manifest to `ACCESS_FINE_LOCATION`, as shown in the following example:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.google.android.gms.location.sample.locationupdates"
>

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

Define an Intent Service to Fetch the Address

The `getFromLocation()` method provided by the `Geocoder` class accepts a latitude and longitude, and returns a list of addresses. The method is synchronous, and may take a long time to do its work, so you should not call it from the main, user interface (UI) thread of your app.

The `IntentService` class provides a structure for running a task on a

background thread. Using this class, you can handle a long-running operation without affecting your UI's responsiveness. Note that the `AsyncTask` class also allows you to perform background operations, but it's designed for short operations. An `AsyncTask` shouldn't keep a reference to the UI if the activity is recreated, for example when the device is rotated. In contrast, an `IntentService` doesn't need to be cancelled when the activity is rebuilt.

Define a `FetchAddressIntentService` class that extends `IntentService`. This class is your address lookup service. The intent service handles an intent asynchronously on a worker thread, and stops itself when it runs out of work. The intent extras provide the data needed by the service, including a `Location` object for conversion to an address, and a `ResultReceiver` object to handle the results of the address lookup. The service uses a `Geocoder` to fetch the address for the location, and sends the results to the `ResultReceiver`.

Define the Intent Service in your App Manifest

Add an entry to your app manifest defining the intent service:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.google.android.gms.location.sample.locationaddress"
>
    <application
        ...
        <service
            android:name=".FetchAddressIntentService"
            android:exported="false"/>
    </application>
    ...
</manifest>
```

Note: The `<service>` element in the manifest doesn't need to include an intent filter, because your main activity creates an explicit intent by specifying the name of the class to use for the intent.

Create a Geocoder

The process of converting a geographic location to an address is called *reverse geocoding*. To perform the main work of the intent service, that is, your reverse geocoding request, implement `onHandleIntent()` within

the `FetchAddressIntentService` class. Create a `Geocoder` object to handle the reverse geocoding.

A locale represents a specific geographical or linguistic region. Locale objects are used to adjust the presentation of information, such as numbers or dates, to suit the conventions in the region represented by the locale. Pass a `Locale` object to the `Geocoder` object, to ensure that the resulting address is localized to the user's geographic region.

```
@Override
protected void onHandleIntent(Intent intent) {
    Geocoder geocoder = new Geocoder(this, Locale.getDefault());
    ...
}
```

Retrieve the street address data

The next step is to retrieve the street address from the geocoder, handle any errors that may occur, and send the results back to the activity that requested the address. To report the results of the geocoding process, you need two numeric constants that indicate success or failure. Define a `Constants` class to contain the values, as shown in this code snippet:

```
public final class Constants {
    public static final int SUCCESS_RESULT = 0;
    public static final int FAILURE_RESULT = 1;
    public static final String PACKAGE_NAME =
        "com.google.android.gms.location.sample.locationaddress";
    public static final String RECEIVER = PACKAGE_NAME +
".RECEIVER";
    public static final String RESULT_DATA_KEY = PACKAGE_NAME +
        ".RESULT_DATA_KEY";
    public static final String LOCATION_DATA_EXTRA = PACKAGE_NAME +
        ".LOCATION_DATA_EXTRA";
}
```

To get a street address corresponding to a geographical location, call `getFromLocation()`, passing it the latitude and longitude from the location object, and the maximum number of addresses you want returned. In this case, you want just one address. The geocoder returns an array of addresses. If no addresses were found to match the given location, it returns an empty list. If there is no backend geocoding service available, the geocoder returns null.

Check for the following errors as shown in the code sample below. If an

error occurs, place the corresponding error message in the `errorMessage` variable, so you can send it back to the requesting activity:

- **No location data provided** - The intent extras do not include the `Location` object required for reverse geocoding.
- **Invalid latitude or longitude used** - The latitude and/or longitude values provided in the `Location` object are invalid.
- **No geocoder available** - The background geocoding service is not available, due to a network error or IO exception.
- **Sorry, no address found** - The geocoder could not find an address for the given latitude/longitude.

To get the individual lines of an address object, use the `getAddressLine()` method provided by the `Address` class. Then join the lines into a list of address fragments ready to return to the activity that requested the address.

To send the results back to the requesting activity, call the `deliverResultToReceiver()` method (defined in Return the address to the requestor). The results consist of the previously-mentioned numeric success/failure code and a string. In the case of a successful reverse geocoding, the string contains the address. In the case of a failure, the string contains the error message, as shown in the code sample below:

```
@Override
protected void onHandleIntent(Intent intent) {
    String errorMessage = "";

    // Get the location passed to this service through an extra.
    Location location = intent.getParcelableExtra(
            Constants.LOCATION_DATA_EXTRA);

    ...

    List<Address> addresses = null;

    try {
        addresses = geocoder.getFromLocation(
                location.getLatitude(),
                location.getLongitude(),
                // In this sample, get just a single address.
                1);
    } catch (IOException ioException) {
        // Catch network or other I/O problems.
        errorMessage = getString(R.string.service_not_available);
        Log.e(TAG, errorMessage, ioException);
    } catch (IllegalArgumentException illegalArgumentException) {
```

```
        // Catch invalid latitude or longitude values.
        errorMessage = getString(R.string.invalid_lat_long_used);
        Log.e(TAG, errorMessage + ". " +
                "Latitude = " + location.getLatitude() +
                ", Longitude = " +
                location.getLongitude(), illegalArgumentException);
    }

    // Handle case where no address was found.
    if (addresses == null || addresses.size()  == 0) {
        if (errorMessage.isEmpty()) {
            errorMessage = getString(R.string.no_address_found);
            Log.e(TAG, errorMessage);
        }
        deliverResultToReceiver(Constants.FAILURE_RESULT,
errorMessage);
    } else {
        Address address = addresses.get(0);
        ArrayList<String> addressFragments = new ArrayList<String>();

        // Fetch the address lines using getAddressLine,
        // join them, and send them to the thread.
        for(int i = 0; i < address.getMaxAddressLineIndex(); i++) {
            addressFragments.add(address.getAddressLine(i));
        }
        Log.i(TAG, getString(R.string.address_found));
        deliverResultToReceiver(Constants.SUCCESS_RESULT,
                TextUtils.join(System.getProperty("line.separator"),
                        addressFragments));
    }
}
```

Return the address to the requestor

The final thing the intent service must do is send the address back to a `ResultReceiver` in the activity that started the service. The `ResultReceiver` class allows you to send a numeric result code as well as a message containing the result data. The numeric code is useful for reporting the success or failure of the geocoding request. In the case of a successful reverse geocoding, the message contains the address. In the case of a failure, the message contains some text describing the reason for failure.

You have already retrieved the address from the geocoder, trapped any errors that may occur, and called the `deliverResultToReceiver()` method. Now you need to define the `deliverResultToReceiver()` method that sends a result code and message bundle to the result receiver.

For the result code, use the value that you've passed to the `deliverResultToReceiver()` method in the `resultCode` parameter. To

construct the message bundle, concatenate the `RESULT_DATA_KEY` constant from your `Constants` class (defined in Retrieve the street address data) and the value in the `message` parameter passed to the `deliverResultToReceiver()` method, as shown in the following sample:

```
public class FetchAddressIntentService extends IntentService {
    protected ResultReceiver mReceiver;
    ...
    private void deliverResultToReceiver(int resultCode, String
message) {
        Bundle bundle = new Bundle();
        bundle.putString(Constants.RESULT_DATA_KEY, message);
        mReceiver.send(resultCode, bundle);
    }
}
```

Start the Intent Service

The intent service, as defined in the previous section, runs in the background and is responsible for fetching the address corresponding to a given geographic location. When you start the service, the Android framework instantiates and starts the service if it isn't already running, and creates a process if needed. If the service is already running then it remains running. Because the service extends `IntentService`, it shuts down automatically when all intents have been processed.

Start the service from your app's main activity, and create an `Intent` to pass data to the service. You need an *explicit* intent, because you want only your service to respond to the intent. For more information, see Intent Types.

To create an explicit intent, specify the name of the class to use for the service: `FetchAddressIntentService.class`. Pass two pieces of information in the intent extras:

- A `ResultReceiver` to handle the results of the address lookup.
- A `Location` object containing the latitude and longitude that you want to convert to an address.

The following code sample shows you how to start the intent service:

```
public class MainActivity extends ActionBarActivity implements
```

```
        ConnectionCallbacks, OnConnectionFailedListener {

    protected Location mLastLocation;
    private AddressResultReceiver mResultReceiver;
    ...

    protected void startIntentService() {
        Intent intent = new Intent(this,
FetchAddressIntentService.class);
        intent.putExtra(Constants.RECEIVER, mResultReceiver);
        intent.putExtra(Constants.LOCATION_DATA_EXTRA,
mLastLocation);
        startService(intent);
    }
}
```

Call the above `startIntentService()` method when the user takes an action that requires a geocoding address lookup. For example, the user may press a *Fetch address* button on your app's UI. Before starting the intent service, you need to check that the connection to Google Play services is present. The following code snippet shows the call to the `startIntentService()` method in the button handler:

```
public void fetchAddressButtonHandler(View view) {
    // Only start the service to fetch the address if
GoogleApiClient is
    // connected.
    if (mGoogleApiClient.isConnected() && mLastLocation != null) {
        startIntentService();
    }
    // If GoogleApiClient isn't connected, process the user's
request by
    // setting mAddressRequested to true. Later, when
GoogleApiClient connects,
    // launch the service to fetch the address. As far as the user
is
    // concerned, pressing the Fetch Address button
    // immediately kicks off the process of getting the address.
    mAddressRequested = true;
    updateUIWidgets();
}
```

You must also start the intent service when the connection to Google Play services is established, if the user has already clicked the button on your app's UI. The following code snippet shows the call to the `startIntentService()` method in the `onConnected()` callback provided by the Google API Client:

```
public class MainActivity extends ActionBarActivity implements
        ConnectionCallbacks, OnConnectionFailedListener {
    ...
    @Override
    public void onConnected(Bundle connectionHint) {
```

```
        // Gets the best and most recent location currently
available,
        // which may be null in rare cases when a location is not
available.
        mLastLocation =
LocationServices.FusedLocationApi.getLastLocation(
                mGoogleApiClient);

        if (mLastLocation != null) {
            // Determine whether a Geocoder is available.
            if (!Geocoder.isPresent()) {
                Toast.makeText(this, R.string.no_geocoder_available,
                        Toast.LENGTH_LONG).show();
                return;
            }

            if (mAddressRequested) {
                startIntentService();
            }
        }
    }
}
```

Receive the Geocoding Results

The intent service has handled the geocoding request, and uses a `ResultReceiver` to return the results to the activity that made the request. In the activity that makes the request, define an `AddressResultReceiver` that extends `ResultReceiver` to handle the response from `FetchAddressIntentService`.

The result includes a numeric result code (`resultCode`) as well as a message containing the result data (`resultData`). If the reverse geocoding process was successful, the `resultData` contains the address. In the case of a failure, the `resultData` contains text describing the reason for failure. For details of the possible errors, see Return the address to the requestor.

Override the `onReceiveResult()` method to handle the results delivered to the result receiver, as shown in the following code sample:

```
public class MainActivity extends ActionBarActivity implements
        ConnectionCallbacks, OnConnectionFailedListener {
    ...
    class AddressResultReceiver extends ResultReceiver {
        public AddressResultReceiver(Handler handler) {
            super(handler);
        }
```

```
        @Override
        protected void onReceiveResult(int resultCode, Bundle
resultData) {

            // Display the address string
            // or an error message sent from the intent service.
            mAddressOutput =
resultData.getString(Constants.RESULT_DATA_KEY);
            displayAddressOutput();

            // Show a toast message if an address was found.
            if (resultCode == Constants.SUCCESS_RESULT) {
                showToast(getString(R.string.address_found));
            }

        }
    }
}
```

Next Get started Next class

Creating and Monitoring Geofences

Previous Next Get started

This lesson teaches you to

1. Set up for Geofence Monitoring
2. Create and Add Geofences
3. Handle Geofence Transitions
4. Stop Geofence Monitoring

You should also read

- Setup Google Play Services SDK

Try it out

- Geofencing

Geofencing combines awareness of the user's current location with awareness of the user's proximity to locations that may be of interest. To mark a location of interest, you specify its latitude and longitude. To adjust the proximity for the location, you add a radius. The latitude, longitude, and radius define a geofence, creating a circular area, or fence, around the location of interest.

You can have multiple active geofences, with a limit of 100 per device user. For each geofence, you can ask Location Services to send you entrance and exit events, or you can specify a duration within the geofence area to wait, or *dwell*, before triggering an event. You can limit the duration of any geofence by specifying an expiration duration in

milliseconds. After the geofence expires, Location Services automatically removes it.

This lesson shows you how to add and remove geofences, and then listen for geofence transitions using an `IntentService`.

Set up for Geofence Monitoring

The first step in requesting geofence monitoring is to request the necessary permission. To use geofencing, your app must request `ACCESS_FINE_LOCATION`. To request this permission, add the following element as a child element of the `<manifest>` element in your app manifest:

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

If you want to use an `IntentService` to listen for geofence transitions, add an element specifying the service name. This element must be a `<application>`

child of the element:

```
<application
   android:allowBackup="true">
   ...
   <service android:name=".GeofenceTransitionsIntentService"/>
<application/>
```

To access the location APIs, you need to create an instance of the Google Play services API client. To learn how to connect your client, see Connect to Google Play Services.

Create and Add Geofences

Your app needs to create and add geofences using the location API's builder class for creating Geofence objects, and the convenience class for adding them. Also, to handle the intents sent from Location Services when geofence transitions occur, you can define a `PendingIntent` as shown in this section.

Create geofence objects

First, use `Geofence.Builder` to create a geofence, setting the desired radius, duration, and transition types for the geofence. For example, to populate a list object named `mGeofenceList`:

```
mGeofenceList.add(new Geofence.Builder()
    // Set the request ID of the geofence. This is a string to
identify this
    // geofence.
    .setRequestId(entry.getKey())

    .setCircularRegion(
            entry.getValue().latitude,
            entry.getValue().longitude,
            Constants.GEOFENCE_RADIUS_IN_METERS
    )

.setExpirationDuration(Constants.GEOFENCE_EXPIRATION_IN_MILLISECONDS)
    .setTransitionTypes(Geofence.GEOFENCE_TRANSITION_ENTER |
            Geofence.GEOFENCE_TRANSITION_EXIT)
    .build());
```

This example pulls data from a constants file. In actual practice, apps might dynamically create geofences based on the user's location.

Specify geofences and initial triggers

The following snippet uses the `GeofencingRequest` class and its nested `GeofencingRequestBuilder` class to specify the geofences to monitor and to set how related geofence events are triggered:

```
private GeofencingRequest getGeofencingRequest() {
    GeofencingRequest.Builder builder = new
GeofencingRequest.Builder();

builder.setInitialTrigger(GeofencingRequest.INITIAL_TRIGGER_ENTER);
    builder.addGeofences(mGeofenceList);
    return builder.build();
}
```

This example shows the use of two geofence triggers. The `GEOFENCE_TRANSITION_ENTER` transition triggers when a device enters a geofence, and the `GEOFENCE_TRANSITION_EXIT` transition triggers when a device exits a geofence. Specifying `INITIAL_TRIGGER_ENTER` tells Location services that `GEOFENCE_TRANSITION_ENTER` should be triggered if the the device is already inside the geofence.

In many cases, it may be preferable to use instead `INITIAL_TRIGGER_DWELL`, which triggers events only when the user stops for a defined duration within a geofence. This approach can help reduce "alert spam" resulting from large numbers notifications when a device briefly enters and exits geofences. Another strategy for getting best results from your geofences is to set a minimum radius of 100 meters. This helps account for the location accuracy of typical WiFi networks, and also helps reduce device power consumption.

Define an Intent for geofence transitions

The `Intent` sent from Location Services can trigger various actions in your app, but you should *not* have it start an activity or fragment, because components should only become visible in response to a user action. In many cases, an `IntentService` is a good way to handle the intent. An `IntentService` can post a notification, do long-running background work, send intents to other services, or send a broadcast intent. The following snippet shows how to define a `PendingIntent` that starts an `IntentService`:

```
public class MainActivity extends FragmentActivity {
    ...
    private PendingIntent getGeofencePendingIntent() {
        // Reuse the PendingIntent if we already have it.
        if (mGeofencePendingIntent != null) {
            return mGeofencePendingIntent;
        }
        Intent intent = new Intent(this,
GeofenceTransitionsIntentService.class);
        // We use FLAG_UPDATE_CURRENT so that we get the same
pending intent back when
        // calling addGeofences() and removeGeofences().
        return PendingIntent.getService(this, 0, intent,
PendingIntent.
                FLAG_UPDATE_CURRENT);
    }
```

Add geofences

To add geofences, use the `GeoencingApi.addGeofences()` method. Provide the Google API client, the `GeofencingRequest` object, and the `PendingIntent`. The following snippet, which processes the results in `onResult()`, assumes that the main activity implements `ResultCallback`:

```
public class MainActivity extends FragmentActivity {
    ...
    LocationServices.GeofencingApi.addGeofences(
                mGoogleApiClient,
                getGeofencingRequest(),
                getGeofencePendingIntent()
        ).setResultCallback(this);
```

Handle Geofence Transitions

When Location Services detects that the user has entered or exited a geofence, it sends out the `Intent` contained in the `PendingIntent` you included in the request to add geofences. This `Intent` is received by a service like `GeofenceTransitionsIntentService`, which obtains the geofencing event from the intent, determines the type of Geofence transition(s), and determines which of the defined geofences was triggered. It then sends a notification as the output.

The following snippet shows how to define an `IntentService` that posts a notification when a geofence transition occurs. When the user clicks the notification, the app's main activity appears:

```
public class GeofenceTransitionsIntentService extends IntentService
{
   ...
    protected void onHandleIntent(Intent intent) {
        GeofencingEvent geofencingEvent =
GeofencingEvent.fromIntent(intent);
        if (geofencingEvent.hasError()) {
            String errorMessage =
GeofenceErrorMessages.getErrorString(this,
                    geofencingEvent.getErrorCode());
            Log.e(TAG, errorMessage);
            return;
        }

        // Get the transition type.
        int geofenceTransition =
geofencingEvent.getGeofenceTransition();

        // Test that the reported transition was of interest.
        if (geofenceTransition == Geofence.GEOFENCE_TRANSITION_ENTER
||
                geofenceTransition ==
Geofence.GEOFENCE_TRANSITION_EXIT) {

            // Get the geofences that were triggered. A single event
can trigger
            // multiple geofences.
            List triggeringGeofences =
geofencingEvent.getTriggeringGeofences();

            // Get the transition details as a String.
            String geofenceTransitionDetails =
getGeofenceTransitionDetails(
                    this,
                    geofenceTransition,
                    triggeringGeofences
            );

            // Send notification and log the transition details.
            sendNotification(geofenceTransitionDetails);
            Log.i(TAG, geofenceTransitionDetails);
        } else {
            // Log the error.
            Log.e(TAG,
getString(R.string.geofence_transition_invalid_type,
                    geofenceTransition));
        }
    }
```

After detecting the transition event via the `PendingIntent`, this `IntentService` gets the geofence transition type and tests whether it is one of the events the app uses to trigger notifications -- either `GEOFENCE_TRANSITION_ENTER` or `GEOFENCE_TRANSITION_EXIT` in this case. The service then sends a notification and logs the transition details.

Stop Geofence Monitoring

Stopping geofence monitoring when it is no longer needed or desired can help save battery power and CPU cycles on the device. You can stop geofence monitoring in the main activity used to add and remove geofences; removing a geofence stops it immediately. The API provides methods to remove geofences either by request IDs, or by removing geofences associated with a given `PendingIntent`.

The following snippet removes geofences by `PendingIntent`, stopping all further notification when the device enters or exits previously added geofences:

```
LocationServices.GeofencingApi.removeGeofences(
            mGoogleApiClient,
            // This is the same pending intent that was used in
addGeofences().
            getGeofencePendingIntent()
    ).setResultCallback(this); // Result processed in onResult().
}
```

You can combine geofencing with other location-aware features, such as periodic location updates. For more information, see the other lessons in this class.

Next Get started Next class

Adding Maps

Allow your users to explore the world with rich maps provided by Google. Identify locations with custom markers, augment the map data with image overlays, embed one or more maps as fragments, and much more.

The Google Maps Android API allows you to include maps and customized mapping information in your app.

Key Developer Features

Add maps to your app

With Google Maps Android API v2, you can embed maps into an activity as a fragment with a simple XML snippet. The new Maps offer exciting features such as 3D maps; indoor, satellite, terrain, and hybrid maps; vector-based tiles for efficient caching and drawing; animated transitions; and much more. Add a map object.

Customize the map

Add markers onto the map to indicate special points of interest for your users. You can define custom colors or icons for your map markers to match your app's look and feel. To further enhance the app, draw polylines and polygons to indicate paths or regions, or provide complete image overlays. Draw markers.

Control the user's view

Give your users a different view of the world with the ability to control the rotation, tilt, zoom, and pan properties of the "camera" perspective of the map. Change the view.

Add Street View to your app

Embed Street View into an activity and let your users explore the world through panoramic 360-degree views. Programmatically control the zoom and orientation (tilt and bearing) of the Street View camera, and animate the camera movements over a given duration. Add Street View.

Get Started

Google Maps Android API is part of the Google Play services platform. To use Google Maps, set up the Google Play services SDK in your app development project. For more information, see the Getting Started guide for the Google Maps Android API.

Building Apps with User Info & Location

These classes teach you how to add user personalization to your app. Some of the ways you can do this is by identifying users, providing information that's relevant to them, and providing information about the world around them.

Accessing Contacts Data

Previous Next Get started

Dependencies and prerequisites

- Android 2.0 (API Level 5) or higher
- Experience in using `Intent` objects
- Experience in using content providers

You should also read

- Content Provider Basics
- Contacts Provider

Try it out

Download the sample

ContactsList.zip

The Contacts Provider is the central repository of the user's contacts information, including data from contacts apps and social networking apps. In your apps, you can access Contacts Provider information directly by calling `ContentResolver` methods or by sending intents to a contacts app.

This class focuses on retrieving lists of contacts, displaying the details for a particular contact, and modifying contacts using intents. The basic techniques described here can be extended to perform more complex tasks. In addition, this class helps you understand the overall structure and operation of the Contacts Provider.

Lessons

Retrieving a List of Contacts

Learn how to retrieve a list of contacts for which the data matches all or part of a search string, using the following techniques:

- Match by contact name
- Match any type of contact data
- Match a specific type of contact data, such as a phone number

Retrieving Details for a Contact

Learn how to retrieve the details for a single contact. A contact's details are data such as phone numbers and email addresses. You can retrieve all details, or you can retrieve details of a specific type, such as all email addresses.

Modifying Contacts Using Intents

Learn how to modify a contact by sending an intent to the People app.

Displaying the Quick Contact Badge

Learn how to display the `QuickContactBadge` widget. When the user clicks the contact badge widget, a dialog opens that displays the contact's details and action buttons for apps that can handle the details. For example, if the contact has an email address, the dialog displays an action button for the default email app.

Next Get started Next class

Retrieving a List of Contacts

Previous Next Get started

This lesson teaches you to

1. Request Permission to Read the Provider
2. Match a Contact by Name and List the Results
3. Match a Contact By a Specific Type of Data
4. Match a Contact By Any Type of Data

You should also read

- Content Provider Basics
- Contacts Provider
- Loaders
- Creating a Search Interface

Try it out

Download the sample

ContactsList.zip

This lesson shows you how to retrieve a list of contacts whose data matches all or part of a search string, using the following techniques:

Match contact names
: Retrieve a list of contacts by matching the search string to all or part of the contact name data. The Contacts Provider allows multiple instances of the same name, so this technique can return a list of matches.

Match a specific type of data, such as a phone number

Retrieve a list of contacts by matching the search string to a particular type of detail data such as an email address. For example, this technique allows you to list all of the contacts whose email address matches the search string.

Match any type of data

Retrieve a list of contacts by matching the search string to any type of detail data, including name, phone number, street address, email address, and so forth. For example, this technique allows you to accept any type of data for a search string and then list the contacts for which the data matches the string.

Note: All the examples in this lesson use a `CursorLoader` to retrieve data from the Contacts Provider. A `CursorLoader` runs its query on a thread that's separate from the UI thread. This ensures that the query doesn't slow down UI response times and cause a poor user experience. For more information, see the Android training class Loading Data in the Background.

Request Permission to Read the Provider

To do any type of search of the Contacts Provider, your app must have `READ_CONTACTS` permission. To request this, add this `<uses-permission>` element to your manifest file as a child element of `<manifest>`:

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

Match a Contact by Name and List the Results

This technique tries to match a search string to the name of a contact or contacts in the Contact Provider's `ContactsContract.Contacts` table. You usually want to display the results in a `ListView`, to allow the user to choose among the matched contacts.

Define ListView and item layouts

To display the search results in a `ListView`, you need a main layout file that defines the entire UI including the `ListView`, and an item layout file that defines one line of the `ListView`. For example, you could create the main layout file `res/layout/contacts_list_view.xml` with the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<ListView xmlns:android="http://schemas.android.com/apk/res/android"
          android:id="@android:id/list"
          android:layout_width="match_parent"
          android:layout_height="match_parent"/>
```

This XML uses the built-in Android `ListView` widget `android:id/list`.

Define the item layout file `contacts_list_item.xml` with the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
          android:id="@android:id/text1"
          android:layout_width="match_parent"
          android:layout_height="wrap_content"
          android:clickable="true"/>
```

This XML uses the built-in Android `TextView` widget `android:text1`.

Note: This lesson doesn't describe the UI for getting a search string from the user, because you may want to get the string indirectly. For example, you can give the user an option to search for contacts whose name matches a string in an incoming text message.

The two layout files you've written define a user interface that shows a `ListView`. The next step is to write code that uses this UI to display a list of contacts.

Define a Fragment that displays the list of contacts

To display the list of contacts, start by defining a `Fragment` that's loaded by an `Activity`. Using a `Fragment` is a more flexible technique, because you can use one `Fragment` to display the list and a second `Fragment` to display the details for a contact that the user chooses from the list. Using this approach, you can combine one of the techniques presented in this lesson with one from the lesson Retrieving Details for a Contact.

To learn how to use one or more `Fragment` objects from an an `Activity`, read the training class Building a Dynamic UI with Fragments.

To help you write queries against the Contacts Provider, the Android framework provides a contracts class called `ContactsContract`, which defines useful constants and methods for accessing the provider. When you use this class, you don't have to define your own constants for content URIs, table names, or columns. To use this class, include the following statement:

```
import android.provider.ContactsContract;
```

Since the code uses a `CursorLoader` to retrieve data from the provider, you must specify that it implements the loader interface `LoaderManager.LoaderCallbacks`. Also, to help detect which contact the user selects from the list of search results, implement the adapter interface `AdapterView.OnItemClickListener`. For example:

```
...
import android.support.v4.app.Fragment;
import android.support.v4.app.LoaderManager.LoaderCallbacks;
import android.widget.AdapterView;
...
public class ContactsFragment extends Fragment implements
        LoaderManager.LoaderCallbacks<Cursor>,
        AdapterView.OnItemClickListener {
```

Define global variables

Define global variables that are used in other parts of the code:

```
    ...
    /*
     * Defines an array that contains column names to move from
     * the Cursor to the ListView.
     */
    @SuppressLint("InlinedApi")
    private final static String[] FROM_COLUMNS = {
            Build.VERSION.SDK_INT
                    >= Build.VERSION_CODES.HONEYCOMB ?
                    Contacts.DISPLAY_NAME_PRIMARY :
                    Contacts.DISPLAY_NAME
    };
    /*
     * Defines an array that contains resource ids for the layout
views
     * that get the Cursor column contents. The id is pre-defined in
```

```
     * the Android framework, so it is prefaced with "android.R.id"
     */
    private final static int[] TO_IDS = {
           android.R.id.text1
    };
    // Define global mutable variables
    // Define a ListView object
    ListView mContactsList;
    // Define variables for the contact the user selects
    // The contact's _ID value
    long mContactId;
    // The contact's LOOKUP_KEY
    String mContactKey;
    // A content URI for the selected contact
    Uri mContactUri;
    // An adapter that binds the result Cursor to the ListView
    private SimpleCursorAdapter mCursorAdapter;
    ...
```

Note: Since `Contacts.DISPLAY_NAME_PRIMARY` requires Android 3.0 (API version 11) or later, setting your app's `minSdkVersion` to 10 or below generates an Android Lint warning in Eclipse with ADK. To turn off this warning, add the annotation `@SuppressLint("InlinedApi")` before the definition of `FROM_COLUMNS`.

Initialize the Fragment

Initialize the `Fragment`. Add the empty, public constructor required by the Android system, and inflate the `Fragment` object's UI in the callback method `onCreateView()`. For example:

```
    // Empty public constructor, required by the system
    public ContactsFragment() {}

    // A UI Fragment must inflate its View
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup
container,
            Bundle savedInstanceState) {
        // Inflate the fragment layout
        return inflater.inflate(R.layout.contact_list_fragment,
            container, false);
    }
```

Set up the CursorAdapter for the ListView

Set up the `SimpleCursorAdapter` that binds the results of the search to the `ListView`. To get the `ListView` object that displays the contacts, you need to call `Activity.findViewById()` using the parent activity of the `Fragment`.

Use the `Context` of the parent activity when you call `setAdapter()`. For example:

```
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        ...
        // Gets the ListView from the View list of the parent
activity
        mContactsList =
            (ListView)
getActivity().findViewById(R.layout.contact_list_view);
        // Gets a CursorAdapter
        mCursorAdapter = new SimpleCursorAdapter(
                getActivity(),
                R.layout.contact_list_item,
                null,
                FROM_COLUMNS, TO_IDS,
                0);
        // Sets the adapter for the ListView
        mContactsList.setAdapter(mCursorAdapter);
    }
```

Set the selected contact listener

When you display the results of a search, you usually want to allow the user to select a single contact for further processing. For example, when the user clicks a contact you can display the contact's address on a map. To provide this feature, you first defined the current `Fragment` as the click listener by specifying that the class implements `AdapterView.OnItemClickListener`, as shown in the section Define a Fragment that displays the list of contacts.

To continue setting up the listener, bind it to the `ListView` by calling the method `setOnItemClickListener()` in `onActivityCreated()`. For example:

```
    public void onActivityCreated(Bundle savedInstanceState) {
        ...
        // Set the item click listener to be the current fragment.
        mContactsList.setOnItemClickListener(this);
        ...
    }
```

Since you specified that the current `Fragment` is the `OnItemClickListener` for the `ListView`, you now need to implement its required method `onItemClick()`, which handles the click event. This is described in a succeeding section.

Define a projection

Define a constant that contains the columns you want to return from your query. Each item in the `ListView` displays the contact's display name, which contains the main form of the contact's name. In Android 3.0 (API version 11) and later, the name of this column is `Contacts.DISPLAY_NAME_PRIMARY`; in versions previous to that, its name is `Contacts.DISPLAY_NAME`.

The column `Contacts._ID` is used by the `SimpleCursorAdapter` binding process. `Contacts._ID` and `LOOKUP_KEY` are used together to construct a content URI for the contact the user selects.

```
...
@SuppressLint("InlinedApi")
private static final String[] PROJECTION =
        {
            Contacts._ID,
            Contacts.LOOKUP_KEY,
            Build.VERSION.SDK_INT
                    >= Build.VERSION_CODES.HONEYCOMB ?
                    Contacts.DISPLAY_NAME_PRIMARY :
                    Contacts.DISPLAY_NAME

        };
```

Define constants for the Cursor column indexes

To get data from an individual column in a `Cursor`, you need the column's index within the `Cursor`. You can define constants for the indexes of the `Cursor` columns, because the indexes are the same as the order of the column names in your projection. For example:

```
// The column index for the _ID column
private static final int CONTACT_ID_INDEX = 0;
// The column index for the LOOKUP_KEY column
private static final int LOOKUP_KEY_INDEX = 1;
```

Specify the selection criteria

To specify the data you want, create a combination of text expressions and variables that tell the provider the data columns to search and the values to find.

For the text expression, define a constant that lists the search columns. Although this expression can contain values as well, the preferred practice is to represent the values with a "?" placeholder. During retrieval, the placeholder is replaced with values from an array. Using "?" as a placeholder ensures that the search specification is generated by binding rather than by SQL compilation. This practice eliminates the possibility of malicious SQL injection. For example:

```
    // Defines the text expression
    @SuppressLint("InlinedApi")
    private static final String SELECTION =
            Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB ?
            Contacts.DISPLAY_NAME_PRIMARY + " LIKE ?" :
            Contacts.DISPLAY_NAME + " LIKE ?";
    // Defines a variable for the search string
    private String mSearchString;
    // Defines the array to hold values that replace the ?
    private String[] mSelectionArgs = { mSearchString };
```

Define the onItemClick() method

In a previous section, you set the item click listener for the `ListView`. Now implement the action for the listener by defining the method `AdapterView.OnItemClickListener.onItemClick()`:

```
    @Override
    public void onItemClick(
        AdapterView<?> parent, View item, int position, long rowID)
{
        // Get the Cursor
        Cursor cursor = parent.getAdapter().getCursor();
        // Move to the selected contact
        cursor.moveToPosition(position);
        // Get the _ID value
        mContactId = getLong(CONTACT_ID_INDEX);
        // Get the selected LOOKUP KEY
        mContactKey = getString(CONTACT_KEY_INDEX);
        // Create the contact's content Uri
        mContactUri = Contacts.getLookupUri(mContactId, mContactKey);
        /*
         * You can use mContactUri as the content URI for retrieving
         * the details for a contact.
         */
    }
```

Initialize the loader

Since you're using a `CursorLoader` to retrieve data, you must initialize the background thread and other variables that control asynchronous

retrieval. Do the initialization in `onActivityCreated()`, which is invoked immediately before the `Fragment` UI appears, as shown in the following example:

```
public class ContactsFragment extends Fragment implements
        LoaderManager.LoaderCallbacks<Cursor> {
    ...
    // Called just before the Fragment displays its UI
    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        // Always call the super method first
        super.onActivityCreated(savedInstanceState);
        ...
        // Initializes the loader
        getLoaderManager().initLoader(0, null, this);
```

Implement onCreateLoader()

Implement the method `onCreateLoader()`, which is called by the loader framework immediately after you call `initLoader()`.

In `onCreateLoader()`, set up the search string pattern. To make a string into a pattern, insert "%" (percent) characters to represent a sequence of zero or more characters, or "_" (underscore) characters to represent a single character, or both. For example, the pattern "%Jefferson%" would match both "Thomas Jefferson" and "Jefferson Davis".

Return a new `CursorLoader` from the method. For the content URI, use `Contacts.CONTENT_URI`. This URI refers to the entire table, as shown in the following example:

```
    ...
    @Override
    public Loader<Cursor> onCreateLoader(int loaderId, Bundle args)
{
        /*
         * Makes search string into pattern and
         * stores it in the selection array
         */
        mSelectionArgs[0] = "%" + mSearchString + "%";
        // Starts the query
        return new CursorLoader(
                getActivity(),
                Contacts.CONTENT_URI,
                PROJECTION,
                SELECTION,
                mSelectionArgs,
                null
        );
```

```
    }
```

Implement onLoadFinished() and onLoaderReset()

Implement the `onLoadFinished()` method. The loader framework calls `onLoadFinished()` when the Contacts Provider returns the results of the query. In this method, put the result `Cursor` in the `SimpleCursorAdapter`. This automatically updates the `ListView` with the search results:

```
    public void onLoadFinished(Loader<Cursor> loader, Cursor cursor)
{
        // Put the result Cursor in the adapter for the ListView
        mCursorAdapter.swapCursor(cursor);
    }
```

The method `onLoaderReset()` is invoked when the loader framework detects that the result `Cursor` contains stale data. Delete the `SimpleCursorAdapter` reference to the existing `Cursor`. If you don't, the loader framework will not recycle the `Cursor`, which causes a memory leak. For example:

```
    @Override
    public void onLoaderReset(Loader<Cursor> loader) {
        // Delete the reference to the existing Cursor
        mCursorAdapter.swapCursor(null);

    }
```

You now have the key pieces of an app that matches a search string to contact names and returns the result in a `ListView`. The user can click a contact name to select it. This triggers a listener, in which you can work further with the contact's data. For example, you can retrieve the contact's details. To learn how to do this, continue with the next lesson, Retrieving Details for a Contact.

To learn more about search user interfaces, read the API guide Creating a Search Interface.

The remaining sections in this lesson demonstrate other ways of finding contacts in the Contacts Provider.

Match a Contact By a Specific Type of Data

This technique allows you to specify the type of data you want to match. Retrieving by name is a specific example of this type of query, but you can also do it for any of the types of detail data associated with a contact. For example, you can retrieve contacts that have a specific postal code; in this case, the search string has to match data stored in a postal code row.

To implement this type of retrieval, first implement the following code, as listed in previous sections:

- Request Permission to Read the Provider.
- Define ListView and item layouts.
- Define a Fragment that displays the list of contacts.
- Define global variables.
- Initialize the Fragment.
- Set up the CursorAdapter for the ListView.
- Set the selected contact listener.
- Define constants for the Cursor column indexes.

  Although you're retrieving data from a different table, the order of the columns in the projection is the same, so you can use the same indexes for the Cursor.

- Define the onItemClick() method.
- Initialize the loader.
- Implement onLoadFinished() and onLoaderReset().

The following steps show you the additional code you need to match a search string to a particular type of detail data and display the results.

Choose the data type and table

To search for a particular type of detail data, you have to know the custom MIME type value for the data type. Each data type has a unique MIME type value defined by a constant `CONTENT_ITEM_TYPE` in the subclass of `ContactsContract.CommonDataKinds` associated with the data type. The subclasses have names that indicate their data type; for example, the subclass for email data is `ContactsContract.CommonDataKinds.Email`, and

the custom MIME type for email data is defined by the constant `Email.CONTENT_ITEM_TYPE`.

Use the `ContactsContract.Data` table for your search. All of the constants you need for your projection, selection clause, and sort order are defined in or inherited by this table.

Define a projection

To define a projection, choose one or more of the columns defined in `ContactsContract.Data` or the classes from which it inherits. The Contacts Provider does an implicit join between `ContactsContract.Data` and other tables before it returns rows. For example:

```
@SuppressLint("InlinedApi")
private static final String[] PROJECTION =
    {
        /*
         * The detail data row ID. To make a ListView work,
         * this column is required.
         */
        Data._ID,
        // The primary display name
        Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB ?
                Data.DISPLAY_NAME_PRIMARY :
                Data.DISPLAY_NAME,
        // The contact's _ID, to construct a content URI
        Data.CONTACT_ID
        // The contact's LOOKUP_KEY, to construct a content URI
        Data.LOOKUP_KEY (a permanent link to the contact
    };
```

Define search criteria

To search for a string within a particular type of data, construct a selection clause from the following:

- The name of the column that contains your search string. This name varies by data type, so you need to find the subclass of `ContactsContract.CommonDataKinds` that corresponds to the data type and then choose the column name from that subclass. For example, to search for email addresses, use the column `Email.ADDRESS`.
- The search string itself, represented as the "?" character in the selection clause.

The name of the column that contains the custom MIME type value. This name is always `Data.MIMETYPE`.

- The custom MIME type value for the data type. As described previously, this is the constant `CONTENT_ITEM_TYPE` in the `ContactsContract.CommonDataKinds` subclass. For example, the MIME type value for email data is `Email.CONTENT_ITEM_TYPE`. Enclose the value in single quotes by concatenating a `"'"` (single quote) character to the start and end of the constant; otherwise, the provider interprets the value as a variable name rather than as a string value. You don't need to use a placeholder for this value, because you're using a constant rather than a user-supplied value.

For example:

```
/*
 * Constructs search criteria from the search string
 * and email MIME type
 */
private static final String SELECTION =
        /*
         * Searches for an email address
         * that matches the search string
         */
        Email.ADDRESS + " LIKE ? " + "AND " +
        /*
         * Searches for a MIME type that matches
         * the value of the constant
         * Email.CONTENT_ITEM_TYPE. Note the
         * single quotes surrounding Email.CONTENT_ITEM_TYPE.
         */
        Data.MIMETYPE + " = '" + Email.CONTENT_ITEM_TYPE + "'";
```

Next, define variables to contain the selection argument:

```
String mSearchString;
String[] mSelectionArgs = { "" };
```

Implement onCreateLoader()

Now that you've specified the data you want and how to find it, define a query in your implementation of `onCreateLoader()`. Return a new `CursorLoader` from this method, using your projection, selection text expression, and selection array as arguments. For a content URI, use `Data.CONTENT_URI`. For example:

```
@Override
```

```
    public Loader<Cursor> onCreateLoader(int loaderId, Bundle args)
{
        // OPTIONAL: Makes search string into pattern
        mSearchString = "%" + mSearchString + "%";
        // Puts the search string into the selection criteria
        mSelectionArgs[0] = mSearchString;
        // Starts the query
        return new CursorLoader(
                getActivity(),
                Data.CONTENT_URI,
                PROJECTION,
                SELECTION,
                mSelectionArgs,
                null
        );
    }
```

These code snippets are the basis of a simple reverse lookup based on a specific type of detail data. This is the best technique to use if your app focuses on a particular type of data, such as emails, and you want allow users to get the names associated with a piece of data.

Match a Contact By Any Type of Data

Retrieving a contact based on any type of data returns contacts if any of their data matches a the search string, including name, email address, postal address, phone number, and so forth. This results in a broad set of search results. For example, if the search string is "Doe", then searching for any data type returns the contact "John Doe"; it also returns contacts who live on "Doe Street".

To implement this type of retrieval, first implement the following code, as listed in previous sections:

- Request Permission to Read the Provider.
- Define ListView and item layouts.
-
- Define a Fragment that displays the list of contacts.
- Define global variables.
- Initialize the Fragment.
- Set up the CursorAdapter for the ListView.
- Set the selected contact listener.
- Define a projection.

Define constants for the Cursor column indexes.

For this type of retrieval, you're using the same table you used in the section Match a Contact by Name and List the Results. Use the same column indexes as well.

- Define the onItemClick() method.
- Initialize the loader.
- Implement onLoadFinished() and onLoaderReset().

The following steps show you the additional code you need to match a search string to any type of data and display the results.

Remove selection criteria

Don't define the `SELECTION` constants or the `mSelectionArgs` variable. These aren't used in this type of retrieval.

Implement onCreateLoader()

Implement the `onCreateLoader()` method, returning a new `CursorLoader`. You don't need to convert the search string into a pattern, because the Contacts Provider does that automatically. Use `Contacts.CONTENT_FILTER_URI` as the base URI, and append your search string to it by calling `Uri.withAppendedPath()`. Using this URI automatically triggers searching for any data type, as shown in the following example:

```
    @Override
    public Loader<Cursor> onCreateLoader(int loaderId, Bundle args)
{
        /*
         * Appends the search string to the base URI. Always
         * encode search strings to ensure they're in proper
         * format.
         */
        Uri contentUri = Uri.withAppendedPath(
                Contacts.CONTENT_FILTER_URI,
                Uri.encode(mSearchString));
        // Starts the query
        return new CursorLoader(
                getActivity(),
                contentUri,
                PROJECTION,
```

```
                null,
                null,
                null
        );
    }
```

These code snippets are the basis of an app that does a broad search of the Contacts Provider. The technique is useful for apps that want to implement functionality similar to the People app's contact list screen.

Next Get started Next class

Retrieving Details for a Contact

Previous Next Get started

This lesson teaches you to

1. Retrieve All Details for a Contact
2. Retrieve Specific Details for a Contact

You should also read

- Content Provider Basics
- Contacts Provider
- Loaders

Try it out

Download the sample

ContactsList.zip

This lesson shows how to retrieve detail data for a contact, such as email addresses, phone numbers, and so forth. It's the details that users are looking for when they retrieve a contact. You can give them all the details for a contact, or only display details of a particular type, such as email addresses.

The steps in this lesson assume that you already have a `ContactsContract.Contacts` row for a contact the user has picked. The Retrieving Contact Names lesson shows how to retrieve a list of contacts.

Retrieve All Details for a Contact

To retrieve all the details for a contact, search the `ContactsContract.Data` table for any rows that contain the contact's `LOOKUP_KEY`. This column is available in the `ContactsContract.Data` table, because the Contacts Provider makes an implicit join between the `ContactsContract.Contacts` table and the `ContactsContract.Data` table. The `LOOKUP_KEY` column is described in more detail in the Retrieving Contact Names lesson.

Note: Retrieving all the details for a contact reduces the performance of a device, because it needs to retrieve all of the columns in the `ContactsContract.Data` table. Consider the performance impact before you use this technique.

Request permissions

To read from the Contacts Provider, your app must have `READ_CONTACTS` permission. To request this permission, add the following child element of `<manifest>` to your manifest file:

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

Set up a projection

Depending on the data type a row contains, it may use only a few columns or many. In addition, the data is in different columns depending on the data type. To ensure you get all the possible columns for all possible data types, you need to add all the column names to your projection. Always retrieve `Data._ID` if you're binding the result `Cursor` to a `ListView`; otherwise, the binding won't work. Also retrieve `Data.MIMETYPE` so you can identify the data type of each row you retrieve. For example:

```
    private static final String PROJECTION =
            {
                Data._ID,
                Data.MIMETYPE,
                Data.DATA1,
                Data.DATA2,
                Data.DATA3,
                Data.DATA4,
                Data.DATA5,
                Data.DATA6,
                Data.DATA7,
```

```
                Data.DATA8,
                Data.DATA9,
                Data.DATA10,
                Data.DATA11,
                Data.DATA12,
                Data.DATA13,
                Data.DATA14,
                Data.DATA15
            };
```

This projection retrieves all the columns for a row in the `ContactsContract.Data` table, using the column names defined in the `ContactsContract.Data` class.

Optionally, you can also use any other column constants defined in or inherited by the `ContactsContract.Data` class. Notice, however, that the columns `SYNC1` through `SYNC4` are meant to be used by sync adapters, so their data is not useful.

Define the selection criteria

Define a constant for your selection clause, an array to hold selection arguments, and a variable to hold the selection value. Use the `Contacts.LOOKUP_KEY` column to find the contact. For example:

```
    // Defines the selection clause
    private static final String SELECTION = Data.LOOKUP_KEY + " = ?
";
    // Defines the array to hold the search criteria
    private String[] mSelectionArgs = { "" };
    /*
     * Defines a variable to contain the selection value. Once you
     * have the Cursor from the Contacts table, and you've selected
     * the desired row, move the row's LOOKUP_KEY value into this
     * variable.
     */
    private String mLookupKey;
```

Using "?" as a placeholder in your selection text expression ensures that the resulting search is generated by binding rather than SQL compilation. This approach eliminates the possibility of malicious SQL injection.

Define the sort order

Define the sort order you want in the resulting `Cursor`. To keep all rows for a particular data type together, sort by `Data.MIMETYPE`. This query

argument groups all email rows together, all phone rows together, and so forth. For example:

```
    /*
     * Defines a string that specifies a sort order of MIME type
     */
    private static final String SORT_ORDER = Data.MIMETYPE;
```

Note: Some data types don't use a subtype, so you can't sort on subtype. Instead, you have to iterate through the returned `Cursor`, determine the data type of the current row, and store data for rows that use a subtype. When you finish reading the cursor, you can then sort each data type by subtype and display the results.

Initialize the Loader

Always do retrievals from the Contacts Provider (and all other content providers) in a background thread. Use the Loader framework defined by the `LoaderManager` class and the `LoaderManager.LoaderCallbacks` interface to do background retrievals.

When you're ready to retrieve the rows, initialize the loader framework by calling `initLoader()`. Pass an integer identifier to the method; this identifier is passed to `LoaderManager.LoaderCallbacks` methods. The identifier helps you use multiple loaders in an app by allowing you to differentiate between them.

The following snippet shows how to initialize the loader framework:

```
public class DetailsFragment extends Fragment implements
        LoaderManager.LoaderCallbacks<Cursor> {
    ...
    // Defines a constant that identifies the loader
    DETAILS_QUERY_ID = 0;
    ...
    /*
     * Invoked when the parent Activity is instantiated
     * and the Fragment's UI is ready. Put final initialization
     * steps here.
     */
    @Override
    onActivityCreated(Bundle savedInstanceState) {
        ...
        // Initializes the loader framework
        getLoaderManager().initLoader(DETAILS_QUERY_ID, null, this);
```

Implement onCreateLoader()

Implement the `onCreateLoader()` method, which is called by the loader framework immediately after you call `initLoader()`. Return a `CursorLoader` from this method. Since you're searching the `ContactsContract.Data` table, use the constant `Data.CONTENT_URI` as the content URI. For example:

```
    @Override
    public Loader<Cursor> onCreateLoader(int loaderId, Bundle args)
{
        // Choose the proper action
        switch (loaderId) {
            case DETAILS_QUERY_ID:
            // Assigns the selection parameter
            mSelectionArgs[0] = mLookupKey;
            // Starts the query
            CursorLoader mLoader =
                    new CursorLoader(
                            getActivity(),
                            Data.CONTENT_URI,
                            PROJECTION,
                            SELECTION,
                            mSelectionArgs,
                            SORT_ORDER
                    );
            ...
    }
```

Implement onLoadFinished() and onLoaderReset()

Implement the `onLoadFinished()` method. The loader framework calls `onLoadFinished()` when the Contacts Provider returns the results of the query. For example:

```
    public void onLoadFinished(Loader<Cursor> loader, Cursor cursor)
{
        switch (loader.getId()) {
            case DETAILS_QUERY_ID:
                    /*
                     * Process the resulting Cursor here.
                     */
                }
                break;
            ...
        }
    }
```

The method `onLoaderReset()` is invoked when the loader framework detects that the data backing the result `Cursor` has changed. At this

point, remove any existing references to the `Cursor` by setting them to null. If you don't, the loader framework won't destroy the old `Cursor`, and you'll get a memory leak. For example:

```
@Override
public void onLoaderReset(Loader<Cursor> loader) {
    switch (loader.getId()) {
        case DETAILS_QUERY_ID:
            /*
             * If you have current references to the Cursor,
             * remove them here.
             */
            }
            break;
}
```

Retrieve Specific Details for a Contact

Retrieving a specific data type for a contact, such as all the emails, follows the same pattern as retrieving all details. These are the only changes you need to make to the code listed in Retrieve All Details for a Contact:

Projection
: Modify your projection to retrieve the columns that are specific to the data type. Also modify the projection to use the column name constants defined in the `ContactsContract.CommonDataKinds` subclass corresponding to the data type.

Selection
: Modify the selection text to search for the `MIMETYPE` value that's specific to your data type.

Sort order
: Since you're only selecting a single detail type, don't group the returned `Cursor` by `Data.MIMETYPE`.

These modifications are described in the following sections.

Define a projection

Define the columns you want to retrieve, using the column name constants in the subclass of `ContactsContract.CommonDataKinds` for the

data type. If you plan to bind your `Cursor` to a `ListView`, be sure to retrieve the `_ID` column. For example, to retrieve email data, define the following projection:

```
private static final String[] PROJECTION =
        {
            Email._ID,
            Email.ADDRESS,
            Email.TYPE,
            Email.LABEL
        };
```

Notice that this projection uses the column names defined in the class `ContactsContract.CommonDataKinds.Email`, instead of the column names defined in the class `ContactsContract.Data`. Using the email-specific column names makes the code more readable.

In the projection, you can also use any of the other columns defined in the `ContactsContract.CommonDataKinds` subclass.

Define selection criteria

Define a search text expression that retrieves rows for a specific contact's `LOOKUP_KEY` and the `Data.MIMETYPE` of the details you want. Enclose the `MIMETYPE` value in single quotes by concatenating a `"'"` (single-quote) character to the start and end of the constant; otherwise, the provider interprets the constant as a variable name rather than as a string value. You don't need to use a placeholder for this value, because you're using a constant rather than a user-supplied value. For example:

```
/*
 * Defines the selection clause. Search for a lookup key
 * and the Email MIME type
 */
private static final String SELECTION =
        Data.LOOKUP_KEY + " = ?" +
        " AND " +
        Data.MIMETYPE + " = " +
        "'" + Email.CONTENT_ITEM_TYPE + "'";
// Defines the array to hold the search criteria
private String[] mSelectionArgs = { "" };
```

Define a sort order

Define a sort order for the returned `Cursor`. Since you're retrieving a

specific data type, omit the sort on `MIMETYPE`. Instead, if the type of detail data you're searching includes a subtype, sort on it. For example, for email data you can sort on `Email.TYPE`:

```
private static final String SORT_ORDER = Email.TYPE + " ASC ";
```

Next Get started Next class

Modifying Contacts Using Intents

Previous Next Get started

This lesson teaches you to

1. Insert a New Contact Using an Intent
2. Edit an Existing Contact Using an Intent
3. Let Users Choose to Insert or Edit Using an Intent

You should also read

- Content Provider Basics
- Contacts Provider
- Intents and Intent Filters

Try it out

Download the sample

ContactsList.zip

This lesson shows you how to use an `Intent` to insert a new contact or modify a contact's data. Instead of accessing the Contacts Provider directly, an `Intent` starts the contacts app, which runs the appropriate `Activity`. For the modification actions described in this lesson, if you send extended data in the `Intent` it's entered into the UI of the `Activity` that is started.

Using an `Intent` to insert or update a single contact is the preferred way of modifying the Contacts Provider, for the following reasons:

- It saves you the time and and effort of developing your own UI and

code.
- It avoids introducing errors caused by modifications that don't follow the Contacts Provider's rules.
- It reduces the number of permissions you need to request. Your app doesn't need permission to write to the Contacts Provider, because it delegates modifications to the contacts app, which already has that permission.

Insert a New Contact Using an Intent

You often want to allow the user to insert a new contact when your app receives new data. For example, a restaurant review app can allow users to add the restaurant as a contact as they're reviewing it. To do this using an intent, create the intent using as much data as you have available, and then send the intent to the contacts app.

Inserting a contact using the contacts app inserts a new *raw* contact into the Contacts Provider's `ContactsContract.RawContacts` table. If necessary, the contacts app prompts users for the account type and account to use when creating the raw contact. The contacts app also notifies users if the raw contact already exists. Users then have option of canceling the insertion, in which case no contact is created. To learn more about raw contacts, see the Contacts Provider API guide.

Create an Intent

To start, create a new `Intent` object with the action `Intents.Insert.ACTION`. Set the MIME type to `RawContacts.CONTENT_TYPE`. For example:

```
...
// Creates a new Intent to insert a contact
Intent intent = new Intent(Intents.Insert.ACTION);
// Sets the MIME type to match the Contacts Provider
intent.setType(ContactsContract.RawContacts.CONTENT_TYPE);
```

If you already have details for the contact, such as a phone number or email address, you can insert them into the intent as extended data. For a key value, use the appropriate constant from `Intents.Insert`. The

contacts app displays the data in its insert screen, allowing users to make further edits and additions.

```
/* Assumes EditText fields in your UI contain an email address
 * and a phone number.
 *
 */
private EditText mEmailAddress = (EditText)
findViewById(R.id.email);
private EditText mPhoneNumber = (EditText) findViewById(R.id.phone);
...
/*
 * Inserts new data into the Intent. This data is passed to the
 * contacts app's Insert screen
 */
// Inserts an email address
intent.putExtra(Intents.Insert.EMAIL, mEmailAddress.getText())
/*
 * In this example, sets the email type to be a work email.
 * You can set other email types as necessary.
 */
      .putExtra(Intents.Insert.EMAIL_TYPE,
CommonDataKinds.Email.TYPE_WORK)
// Inserts a phone number
      .putExtra(Intents.Insert.PHONE, mPhoneNumber.getText())
/*
 * In this example, sets the phone type to be a work phone.
 * You can set other phone types as necessary.
 */
      .putExtra(Intents.Insert.PHONE_TYPE, Phone.TYPE_WORK);
```

Once you've created the `Intent`, send it by calling `startActivity()`.

```
    /* Sends the Intent
     */
    startActivity(intent);
```

This call opens a screen in the contacts app that allows users to enter a new contact. The account type and account name for the contact is listed at the top of the screen. Once users enter the data and click *Done*, the contacts app's contact list appears. Users return to your app by clicking *Back*.

Edit an Existing Contact Using an Intent

Editing an existing contact using an `Intent` is useful if the user has already chosen a contact of interest. For example, an app that finds contacts that have postal addresses but lack a postal code could give

users the option of looking up the code and then adding it to the contact.

To edit an existing contact using an intent, use a procedure similar to inserting a contact. Create an intent as described in the section Insert a New Contact Using an Intent, but add the contact's `Contacts.CONTENT_LOOKUP_URI` and the MIME type `Contacts.CONTENT_ITEM_TYPE` to the intent. If you want to edit the contact with details you already have, you can put them in the intent's extended data. Notice that some name columns can't be edited using an intent; these columns are listed in the summary section of the API reference for the class `ContactsContract.Contacts` under the heading "Update".

Finally, send the intent. In response, the contacts app displays an edit screen. When the user finishes editing and saves the edits, the contacts app displays a contact list. When the user clicks *Back*, your app is displayed.

Contacts Lookup Key

A contact's `LOOKUP_KEY` value is the identifier that you should use to retrieve a contact. It remains constant, even if the provider changes the contact's row ID to handle internal operations.

Create the Intent

To edit a contact, call `Intent(action)` to create an intent with the action `ACTION_EDIT`. Call `setDataAndType()` to set the data value for the intent to the contact's `Contacts.CONTENT_LOOKUP_URI` and the MIME type to `Contacts.CONTENT_ITEM_TYPE` MIME type; because a call to `setType()` overwrites the current data value for the `Intent`, you must set the data and the MIME type at the same time.

To get a contact's `Contacts.CONTENT_LOOKUP_URI`, call `Contacts.getLookupUri(id, lookupkey)` with the contact's `Contacts._ID` and `Contacts.LOOKUP_KEY` values as arguments.

The following snippet shows you how to create an intent:

```
    // The Cursor that contains the Contact row
    public Cursor mCursor;
    // The index of the lookup key column in the cursor
    public int mLookupKeyIndex;
    // The index of the contact's _ID value
    public int mIdIndex;
    // The lookup key from the Cursor
    public String mCurrentLookupKey;
    // The _ID value from the Cursor
    public long mCurrentId;
    // A content URI pointing to the contact
    Uri mSelectedContactUri;
    ...
    /*
     * Once the user has selected a contact to edit,
     * this gets the contact's lookup key and _ID values from the
     * cursor and creates the necessary URI.
     */
    // Gets the lookup key column index
    mLookupKeyIndex = mCursor.getColumnIndex(Contacts.LOOKUP_KEY);
    // Gets the lookup key value
    mCurrentLookupKey = mCursor.getString(mLookupKeyIndex);
    // Gets the _ID column index
    mIdIndex = mCursor.getColumnIndex(Contacts._ID);
    mCurrentId = mCursor.getLong(mIdIndex);
    mSelectedContactUri =
            Contacts.getLookupUri(mCurrentId, mCurrentLookupKey);
    ...
    // Creates a new Intent to edit a contact
    Intent editIntent = new Intent(Intent.ACTION_EDIT);
    /*
     * Sets the contact URI to edit, and the data type that the
     * Intent must match
     */

editIntent.setDataAndType(mSelectedContactUri,Contacts.CONTENT_ITEM_TYP
```

Add the navigation flag

In Android 4.0 (API version 14) and later, a problem in the contacts app causes incorrect navigation. When your app sends an edit intent to the contacts app, and users edit and save a contact, when they click *Back* they see the contacts list screen. To navigate back to your app, they have to click *Recents* and choose your app.

To work around this problem in Android 4.0.3 (API version 15) and later, add the extended data key `finishActivityOnSaveCompleted` to the intent, with a value of `true`. Android versions prior to Android 4.0 accept this key, but it has no effect. To set the extended data, do the following:

```
    // Sets the special extended data for navigation
    editIntent.putExtra("finishActivityOnSaveCompleted", true);
```

Add other extended data

To add additional extended data to the `Intent`, call `putExtra()` as desired. You can add extended data for common contact fields by using the key values specified in `Intents.Insert`. Remember that some columns in the `ContactsContract.Contacts` table can't be modified. These columns are listed in the summary section of the API reference for the class `ContactsContract.Contacts` under the heading "Update".

Send the Intent

Finally, send the intent you've constructed. For example:

```
    // Sends the Intent
    startActivity(editIntent);
```

Let Users Choose to Insert or Edit Using an Intent

You can allow users to choose whether to insert a contact or edit an existing one by sending an `Intent` with the action `ACTION_INSERT_OR_EDIT`. For example, an email client app could allow users to add an incoming email address to a new contact, or add it as an additional address for an existing contact. Set the MIME type for this intent to `Contacts.CONTENT_ITEM_TYPE`, but don't set the data URI.

When you send this intent, the contacts app displays a list of contacts. Users can either insert a new contact or pick an existing contact and edit it. Any extended data fields you add to the intent populates the screen that appears. You can use any of the key values specified in `Intents.Insert`. The following code snippet shows how to construct and send the intent:

```
    // Creates a new Intent to insert or edit a contact
    Intent intentInsertEdit = new
Intent(Intent.ACTION_INSERT_OR_EDIT);
    // Sets the MIME type
    intentInsertEdit.setType(Contacts.CONTENT_ITEM_TYPE);
    // Add code here to insert extended data, if desired
```

```
    ...
    // Sends the Intent with an request ID
    startActivity(intentInsertEdit);
```

Next Get started Next class

Displaying the Quick Contact Badge

Previous Next Get started

This lesson teaches you to

1. Add a QuickContactBadge View
2. Set the Contact URI and Thumbnail
3. Add a QuickContactBadge to a ListView

You should also read

- Content Provider Basics
- Contacts Provider

Try it out

Download the sample

ContactsList.zip

This lesson shows you how to add a `QuickContactBadge` to your UI and how to bind data to it. A `QuickContactBadge` is a widget that initially appears as a thumbnail image. Although you can use any `Bitmap` for the thumbnail image, you usually use a `Bitmap` decoded from the contact's photo thumbnail image.

The small image acts as a control; when users click on the image, the `QuickContactBadge` expands into a dialog containing the following:

A large image

The large image associated with the contact, or no image is available, a placeholder graphic.

App icons

An app icon for each piece of detail data that can be handled by a built-in app. For example, if the contact's details include one or more email addresses, an email icon appears. When users click the icon, all of the contact's email addresses appear. When users click one of the addresses, the email app displays a screen for composing a message to the selected email address.

The `QuickContactBadge` view provides instant access to a contact's details, as well as a fast way of communicating with the contact. Users don't have to look up a contact, find and copy information, and then paste it into the appropriate app. Instead, they can click on the `QuickContactBadge`, choose the communication method they want to use, and send the information for that method directly to the appropriate app.

Add a QuickContactBadge View

To add a `QuickContactBadge`, insert a `<QuickContactBadge>` element in your layout. For example:

```
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
                 android:layout_width="match_parent"
                 android:layout_height="match_parent">
...
    <QuickContactBadge
                android:id=@+id/quickbadge
                android:layout_height="wrap_content"
                android:layout_width="wrap_content"
                android:scaleType="centerCrop"/>
    ...
</RelativeLayout>
```

Retrieve provider data

To display a contact in the `QuickContactBadge`, you need a content URI for the contact and a `Bitmap` for the small image. You generate both the content URI and the `Bitmap` from columns retrieved from the Contacts Provider. Specify these columns as part of the projection you use to load

data into your `Cursor`.

For Android 3.0 (API level 11) and later, include the following columns in your projection:

- `Contacts._ID`
- `Contacts.LOOKUP_KEY`
- `Contacts.PHOTO_THUMBNAIL_URI`

For Android 2.3.3 (API level 10) and earlier, use the following columns:

- `Contacts._ID`
- `Contacts.LOOKUP_KEY`

The remainder of this lesson assumes that you've already loaded a `Cursor` that contains these columns as well as others you may have chosen. To learn how to retrieve this columns in a `Cursor`, read the lesson Retrieving a List of Contacts.

Set the Contact URI and Thumbnail

Once you have the necessary columns, you can bind data to the `QuickContactBadge`.

Set the Contact URI

To set the content URI for the contact, call `getLookupUri(id,lookupKey)` to get a `CONTENT_LOOKUP_URI`, then call `assignContactUri()` to set the contact. For example:

```
// The Cursor that contains contact rows
Cursor mCursor;
// The index of the _ID column in the Cursor
int mIdColumn;
// The index of the LOOKUP_KEY column in the Cursor
int mLookupKeyColumn;
// A content URI for the desired contact
Uri mContactUri;
// A handle to the QuickContactBadge view
QuickContactBadge mBadge;
...
mBadge = (QuickContactBadge) findViewById(R.id.quickbadge);
/*
```

```
     * Insert code here to move to the desired cursor row
     */
    // Gets the _ID column index
    mIdColumn = mCursor.getColumnIndex(Contacts._ID);
    // Gets the LOOKUP_KEY index
    mLookupKeyColumn = mCursor.getColumnIndex(Contacts.LOOKUP_KEY);
    // Gets a content URI for the contact
    mContactUri =
            Contacts.getLookupUri(
                mCursor.getLong(mIdColumn),
                mCursor.getString(mLookupKeyColumn)
            );
    mBadge.assignContactUri(mContactUri);
```

When users click the `QuickContactBadge` icon, the contact's details automatically appear in the dialog.

Set the photo thumbnail

Setting the contact URI for the `QuickContactBadge` does not automatically load the contact's thumbnail photo. To load the photo, get a URI for the photo from the contact's `Cursor` row, use it to open the file containing the compressed thumbnail photo, and read the file into a `Bitmap`.

Note: The `PHOTO_THUMBNAIL_URI` column isn't available in platform versions prior to 3.0. For those versions, you must retrieve the URI from the `Contacts.Photo` subtable.

First, set up variables for accessing the `Cursor` containing the `Contacts._ID` and `Contacts.LOOKUP_KEY` columns, as described previously:

```
    // The column in which to find the thumbnail ID
    int mThumbnailColumn;
    /*
     * The thumbnail URI, expressed as a String.
     * Contacts Provider stores URIs as String values.
     */
    String mThumbnailUri;
    ...
    /*
     * Gets the photo thumbnail column index if
     * platform version >= Honeycomb
     */
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
        mThumbnailColumn =
                mCursor.getColumnIndex(Contacts.PHOTO_THUMBNAIL_URI);
    // Otherwise, sets the thumbnail column to the _ID column
    } else {
        mThumbnailColumn = mIdColumn;
    }
    /*
```

```
     * Assuming the current Cursor position is the contact you want,
     * gets the thumbnail ID
     */
    mThumbnailUri = mCursor.getString(mThumbnailColumn);
    ...
```

Define a method that takes photo-related data for the contact and dimensions for the destination view, and returns the properly-sized thumbnail in a `Bitmap`. Start by constructing a URI that points to the thumbnail:

```
    /**
     * Load a contact photo thumbnail and return it as a Bitmap,
     * resizing the image to the provided image dimensions as
needed.
     * @param photoData photo ID Prior to Honeycomb, the contact's
_ID value.
     * For Honeycomb and later, the value of PHOTO_THUMBNAIL_URI.
     * @return A thumbnail Bitmap, sized to the provided width and
height.
     * Returns null if the thumbnail is not found.
     */
    private Bitmap loadContactPhotoThumbnail(String photoData) {
        // Creates an asset file descriptor for the thumbnail file.
        AssetFileDescriptor afd = null;
        // try-catch block for file not found
        try {
            // Creates a holder for the URI.
            Uri thumbUri;
            // If Android 3.0 or later
            if (Build.VERSION.SDK_INT
                    >=
                Build.VERSION_CODES.HONEYCOMB) {
                // Sets the URI from the incoming
PHOTO_THUMBNAIL_URI
                thumbUri = Uri.parse(photoData);
            } else {
            // Prior to Android 3.0, constructs a photo Uri using
_ID
                /*
                 * Creates a contact URI from the Contacts content
URI
                 * incoming photoData (_ID)
                 */
                final Uri contactUri = Uri.withAppendedPath(
                        Contacts.CONTENT_URI, photoData);
                /*
                 * Creates a photo URI by appending the content URI
of
                 * Contacts.Photo.
                 */
                thumbUri =
                        Uri.withAppendedPath(
                                contactUri, Photo.CONTENT_DIRECTORY);
            }

        /*
         * Retrieves an AssetFileDescriptor object for the thumbnail
```

```
         * URI
         * using ContentResolver.openAssetFileDescriptor
         */
        afd = getActivity().getContentResolver().
                openAssetFileDescriptor(thumbUri, "r");
        /*
         * Gets a file descriptor from the asset file descriptor.
         * This object can be used across processes.
         */
        FileDescriptor fileDescriptor = afd.getFileDescriptor();
        // Decode the photo file and return the result as a Bitmap
        // If the file descriptor is valid
        if (fileDescriptor != null) {
            // Decodes the bitmap
            return BitmapFactory.decodeFileDescriptor(
                    fileDescriptor, null, null);
            }
        // If the file isn't found
        } catch (FileNotFoundException e) {
            /*
             * Handle file not found errors
             */
        }
        // In all cases, close the asset file descriptor
        } finally {
            if (afd != null) {
                try {
                    afd.close();
                } catch (IOException e) {}
            }
        }
        return null;
    }
```

Call the `loadContactPhotoThumbnail()` method in your code to get the thumbnail `Bitmap`, and use the result to set the photo thumbnail in your `QuickContactBadge`:

```
    ...
    /*
     * Decodes the thumbnail file to a Bitmap.
     */
    Bitmap mThumbnail =
            loadContactPhotoThumbnail(mThumbnailUri);
    /*
     * Sets the image in the QuickContactBadge
     * QuickContactBadge inherits from ImageView, so
     */
    mBadge.setImageBitmap(mThumbnail);
```

Add a QuickContactBadge to a ListView

A `QuickContactBadge` is a useful addition to a `ListView` that displays a list of contacts. Use the `QuickContactBadge` to display a thumbnail photo for

each contact; when users click the thumbnail, the `QuickContactBadge` dialog appears.

Add the QuickContactBadge element

To start, add a `QuickContactBadge` view element to your item layout For example, if you want to display a `QuickContactBadge` and a name for each contact you retrieve, put the following XML into a layout file:

```
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
                android:layout_width="match_parent"
                android:layout_height="wrap_content">
    <QuickContactBadge
        android:id="@+id/quickcontact"
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:scaleType="centerCrop"/>
    <TextView android:id="@+id/displayname"
              android:layout_width="match_parent"
              android:layout_height="wrap_content"
              android:layout_toRightOf="@+id/quickcontact"
              android:gravity="center_vertical"
              android:layout_alignParentRight="true"
              android:layout_alignParentTop="true"/>
</RelativeLayout>
```

In the following sections, this file is referred to as `contact_item_layout.xml`.

Set up a custom CursorAdapter

To bind a `CursorAdapter` to a `ListView` containing a `QuickContactBadge`, define a custom adapter that extends `CursorAdapter`. This approach allows you to process the data in the `Cursor` before you bind it to the `QuickContactBadge`. This approach also allows you to bind multiple `Cursor` columns to the `QuickContactBadge`. Neither of these operations is possible in a regular `CursorAdapter`.

The subclass of `CursorAdapter` that you define must override the following methods:

`CursorAdapter.newView()`
: Inflates a new `View` object to hold the item layout. In the override of

this method, store handles to the child `View` objects of the layout, including the child `QuickContactBadge`. By taking this approach, you avoid having to get handles to the child `View` objects each time you inflate a new layout.

You must override this method so you can get handles to the individual child `View` objects. This technique allows you to control their binding in `CursorAdapter.bindView()`.

`CursorAdapter.bindView()`

Moves data from the current `Cursor` row to the child `View` objects of the item layout. You must override this method so you can bind both the contact's URI and thumbnail to the `QuickContactBadge`. The default implementation only allows a 1-to-1 mapping between a column and a `View`

The following code snippet contains an example of a custom subclass of `CursorAdapter`:

Define the custom list adapter

Define the subclass of `CursorAdapter` including its constructor, and override `newView()` and `bindView()`:

```
    /**
     *
     *
     */
    private class ContactsAdapter extends CursorAdapter {
        private LayoutInflater mInflater;
        ...
        public ContactsAdapter(Context context) {
            super(context, null, 0);

            /*
             * Gets an inflater that can instantiate
             * the ListView layout from the file.
             */
            mInflater = LayoutInflater.from(context);
            ...
        }
        ...
        /**
         * Defines a class that hold resource IDs of each item
layout
         * row to prevent having to look them up each time data is
         * bound to a row.
```

```
         */
        private class ViewHolder {
            TextView displayname;
            QuickContactBadge quickcontact;
        }
        ..
        @Override
        public View newView(
                Context context,
                Cursor cursor,
                ViewGroup viewGroup) {
            /* Inflates the item layout. Stores resource IDs in a
             * in a ViewHolder class to prevent having to look
             * them up each time bindView() is called.
             */
            final View itemView =
                    mInflater.inflate(
                            R.layout.contact_list_layout,
                            viewGroup,
                            false
                    );
            final ViewHolder holder = new ViewHolder();
            holder.displayname =
                    (TextView) view.findViewById(R.id.displayname);
            holder.quickcontact =
                    (QuickContactBadge)
                            view.findViewById(R.id.quickcontact);
            view.setTag(holder);
            return view;
        }
        ...
        @Override
        public void bindView(
                View view,
                Context context,
                Cursor cursor) {
            final ViewHolder holder = (ViewHolder) view.getTag();
            final String photoData =
                    cursor.getString(mPhotoDataIndex);
            final String displayName =
                    cursor.getString(mDisplayNameIndex);
            ...
            // Sets the display name in the layout
            holder.displayname = cursor.getString(mDisplayNameIndex);
            ...
            /*
             * Generates a contact URI for the QuickContactBadge.
             */
            final Uri contactUri = Contacts.getLookupUri(
                    cursor.getLong(mIdIndex),
                    cursor.getString(mLookupKeyIndex));
            holder.quickcontact.assignContactUri(contactUri);
            String photoData = cursor.getString(mPhotoDataIndex);
            /*
             * Decodes the thumbnail file to a Bitmap.
             * The method loadContactPhotoThumbnail() is defined
             * in the section "Set the Contact URI and Thumbnail"
             */
            Bitmap thumbnailBitmap =
                    loadContactPhotoThumbnail(photoData);
            /*
```

```
             * Sets the image in the QuickContactBadge
             * QuickContactBadge inherits from ImageView
             */
            holder.quickcontact.setImageBitmap(thumbnailBitmap);
    }
```

Set up variables

In your code, set up variables, including a `Cursor` projection that includes the necessary columns.

Note: The following code snippets use the method `loadContactPhotoThumbnail()`, which is defined in the section Set the Contact URI and Thumbnail

For example:

```
public class ContactsFragment extends Fragment implements
        LoaderManager.LoaderCallbacks<Cursor> {
...
    // Defines a ListView
    private ListView mListView;
    // Defines a ContactsAdapter
    private ContactsAdapter mAdapter;
    ...
    // Defines a Cursor to contain the retrieved data
    private Cursor mCursor;
    /*
     * Defines a projection based on platform version. This ensures
     * that you retrieve the correct columns.
     */
    private static final String[] PROJECTION =
            {
                Contacts._ID,
                Contacts.LOOKUP_KEY,
                (Build.VERSION.SDK_INT >=
                 Build.VERSION_CODES.HONEYCOMB) ?
                        Contacts.DISPLAY_NAME_PRIMARY :
                        Contacts.DISPLAY_NAME
                (Build.VERSION.SDK_INT >=
                 Build.VERSION_CODES.HONEYCOMB) ?
                        Contacts.PHOTO_THUMBNAIL_ID :
                        /*
                         * Although it's not necessary to include
the
                         * column twice, this keeps the number of
                         * columns the same regardless of version
                         */
                        Contacts_ID
                ...
            };
    /*
     * As a shortcut, defines constants for the
     * column indexes in the Cursor. The index is
```

```
     * 0-based and always matches the column order
     * in the projection.
     */
    // Column index of the _ID column
    private int mIdIndex = 0;
    // Column index of the LOOKUP_KEY column
    private int mLookupKeyIndex = 1;
    // Column index of the display name column
    private int mDisplayNameIndex = 3;
    /*
     * Column index of the photo data column.
     * It's PHOTO_THUMBNAIL_URI for Honeycomb and later,
     * and _ID for previous versions.
     */
    private int mPhotoDataIndex =
            Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB ?
            3 :
            0;
    ...
```

Set up the ListView

In `Fragment.onCreate()`, instantiate the custom cursor adapter and get a handle to the `ListView`:

```
    @Override
    public void onCreate(Bundle savedInstanceState) {
        ...
        /*
         * Instantiates the subclass of
         * CursorAdapter
         */
        ContactsAdapter mContactsAdapter =
                new ContactsAdapter(getActivity());
        /*
         * Gets a handle to the ListView in the file
         * contact_list_layout.xml
         */
        mListView = (ListView)
findViewById(R.layout.contact_list_layout);
        ...
    }
    ...
```

In `onActivityCreated()`, bind the `ContactsAdapter` to the `ListView`:

```
    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        ...
        // Sets up the adapter for the ListView
        mListView.setAdapter(mAdapter);
        ...
    }
    ...
```

When you get back a `Cursor` containing the contacts data, usually in `onLoadFinished()`, call `swapCursor()` to move the `Cursor` data to the `ListView`. This displays the `QuickContactBadge` for each entry in the list of contacts:

```
    public void onLoadFinished(Loader<Cursor> loader, Cursor cursor)
{
        // When the loader has completed, swap the cursor into the
adapter.
        mContactsAdapter.swapCursor(cursor);
    }
```

When you bind a `Cursor` to a `ListView` with a `CursorAdapter` (or subclass), and you use a `CursorLoader` to load the `Cursor`, always clear references to the `Cursor` in your implementation of `onLoaderReset()`. For example:

```
    @Override
    public void onLoaderReset(Loader<Cursor> loader) {
        // Removes remaining reference to the previous Cursor
        mContactsAdapter.swapCursor(null);
    }
```

Next Get started Next class

Adding Sign-In

The Google+ platform for Android lets you authenticate a user with the same credentials they use on Google every day. Once a user signs in with Google, you can create more engaging experiences and drive usage of your app.

The Google+ Android API allows you to integrate sign-in and social features into your app.

Key Developer Features

Trusted authentication

Google+ Sign-In is a simple, trusted, and secure way to let people sign in to your app with their Google credentials and bring along their Google+ info.
Add sign-in.

Access the profile and social graph

Once users have signed in with Google, your app can welcome them by name, display their picture, connect them with friends, and lots more.
Access the social graph.

Stand out in the stream

Interactive posts is a rich way of sharing to Google+. It lets users prompt friends to take specific actions in your app from a Google+ post, like

"listen," "RSVP," "check-in," and over 100 more actions.
Post interactive content.

Recommend content

Add a native +1 button so users can recommend content from your app. These endorsements can give your app more credibility and help it grow faster.
Add the +1 button.

Get Started

The Google+ Android APIs are part of the Google Play services platform. To use Google+ features, set up the Google Play services SDK in your app development project. For more information, see the Getting Started guide for the Google+ Platform for Android

Building Apps for Wearables

These classes teach you how to build notifications in a handheld app that are automatically synced to wearables as well as how to build apps that run on wearables.

Note: For more information about the APIs used in these training classes, see the Wear API reference documentation.

Adding Wearable Features to Notifications

Previous Next Get started

Dependencies and prerequisites

- Android 4.3 (API Level 18) or higher on the handset device
- An Android Wear device or Wear AVD
- The Android v4 support library (or v13, which includes v4)

You should also read

- Android Wear Design Principles

When an Android handheld (phone or tablet) and Android wearable are connected, the handheld automatically shares notifications with the wearable. On the wearable, each notification appears as a new card in the context stream.

However, to give users the best experience, you should add wearable-specific functionality to the notifications you already create. The following lessons show you how to create notifications that are catered for handhelds and wearables at the same time.

Figure 1. The same notification displayed on a handheld and wearable.

Lessons

Creating a notification
: Learn how to create notifications with wearable features with the Android support library.

Receiving Voice Input in a Notification
: Learn how to add an action to a wearable notification that receives voice input from users and delivers the transcribed message to your handset app.

Adding Pages to a Notification
: Learn how to add additional pages of information that are visible when the user swipes to the left.

Stacking Notifications
: Learn how to place all similar notifications from your app in a stack, allowing users to view each notification individually without adding multiple cards to the card stream.

Next Get started Next class

Creating a Notification for Wearables

Previous Next Get started

This lesson teaches you to

1. Import the Necessary Classes
2. Create Notifications with the Notification Builder
3. Add Action Buttons
4. Specify Wearable-only Actions
5. Add a Big View
6. Add Wearable Features for a Notification
7. Deliver the Notification

To build handheld notifications that are also sent to wearables, use `NotificationCompat.Builder`. When you build notifications with this class, the system takes care of displaying notifications properly, whether they appear on a handheld or wearable.

Note: Notifications using `RemoteViews` are stripped of custom layouts and the wearable only displays the text and icons. However, you can create create custom notifications that use custom card layouts by creating a wearable app that runs on the wearable device.

Import the necessary classes

To import the necessary packages, add this line to your `build.gradle`file:

```
compile "com.android.support:support-v4:20.0.+"
```

Now that your project has access to the necessary packages, import the necessary classes from the support library:

```
import android.support.v4.app.NotificationCompat;
import android.support.v4.app.NotificationManagerCompat;
import android.support.v4.app.NotificationCompat.WearableExtender;
```

Create Notifications with the Notification Builder

The v4 support library allows you to create notifications using the latest notification features such as action buttons and large icons, while remaining compatible with Android 1.6 (API level 4) and higher.

To create a notification with the support library, you create an instance of `NotificationCompat.Builder` and issue the notification by passing it to `notify()`. For example:

```
int notificationId = 001;
// Build intent for notification content
Intent viewIntent = new Intent(this, ViewEventActivity.class);
viewIntent.putExtra(EXTRA_EVENT_ID, eventId);
PendingIntent viewPendingIntent =
        PendingIntent.getActivity(this, 0, viewIntent, 0);

NotificationCompat.Builder notificationBuilder =
        new NotificationCompat.Builder(this)
        .setSmallIcon(R.drawable.ic_event)
        .setContentTitle(eventTitle)
        .setContentText(eventLocation)
        .setContentIntent(viewPendingIntent);

// Get an instance of the NotificationManager service
NotificationManagerCompat notificationManager =
        NotificationManagerCompat.from(this);

// Build the notification and issues it with notification manager.
notificationManager.notify(notificationId,
notificationBuilder.build());
```

When this notification appears on a handheld device, the user can invoke the `PendingIntent` specified by the `setContentIntent()` method by touching the notification. When this notification appears on an Android wearable, the user can swipe the notification to the left to reveal the **Open** action, which invokes the intent on the handheld device.

Add Action Buttons

In addition to the primary content action defined by

`setContentIntent()`, you can add other actions by passing a `PendingIntent` to the `addAction()` method.

For example, the following code shows the same type of notification from above, but adds an action to view the event location on a map.

```
// Build an intent for an action to view a map
Intent mapIntent = new Intent(Intent.ACTION_VIEW);
Uri geoUri = Uri.parse("geo:0,0?q=" + Uri.encode(location));
mapIntent.setData(geoUri);
PendingIntent mapPendingIntent =
        PendingIntent.getActivity(this, 0, mapIntent, 0);

NotificationCompat.Builder notificationBuilder =
        new NotificationCompat.Builder(this)
        .setSmallIcon(R.drawable.ic_event)
        .setContentTitle(eventTitle)
        .setContentText(eventLocation)
        .setContentIntent(viewPendingIntent)
        .addAction(R.drawable.ic_map,
                getString(R.string.map), mapPendingIntent);
```

On a handheld, the action appears as an additional button attached to the notification. On a wearable, the action appears as a large button when the user swipes the notification to the left. When the user taps the action, the associated intent is invoked on the handheld.

Tip: If your notifications include a "Reply" action (such as for a messaging app), you can enhance the behavior by enabling voice input replies directly from the Android wearable. For more information, read Receiving Voice Input from a Notification.

Specify Wearable-only Actions

If you want the actions available on the wearable to be different from those on the handheld, then use `WearableExtender.addAction()`. Once you add an action with this method, the wearable does not display any other actions added with `NotificationCompat.Builder.addAction()`. That is, only the actions added with `WearableExtender.addAction()` appear on the

wearable and they do not appear on the handheld.

```
// Create an intent for the reply action
Intent actionIntent = new Intent(this, ActionActivity.class);
PendingIntent actionPendingIntent =
        PendingIntent.getActivity(this, 0, actionIntent,
                PendingIntent.FLAG_UPDATE_CURRENT);

// Create the action
NotificationCompat.Action action =
        new NotificationCompat.Action.Builder(R.drawable.ic_action,
                getString(R.string.label), actionPendingIntent)
                .build();

// Build the notification and add the action via WearableExtender
Notification notification =
        new NotificationCompat.Builder(mContext)
                .setSmallIcon(R.drawable.ic_message)
                .setContentTitle(getString(R.string.title))
                .setContentText(getString(R.string.content))
                .extend(new WearableExtender().addAction(action))
                .build();
```

Add a Big View

You can insert extended text content to your notification by adding one of the "big view" styles to your notification. On a handheld device, users can see the big view content by expanding the notification. On a wearable device, the big view content is visible by default.

To add the extended content to your notification, call `setStyle()` on the `NotificationCompat.Builder` object, passing it an instance of either `BigTextStyle` or `InboxStyle`.

For example, the following code adds an instance of `NotificationCompat.BigTextStyle` to the event notification, in order to include the complete event description (which includes more text than can fit into the space provided for `setContentText()`).

```
// Specify the 'big view' content to display the long
// event description that may not fit the normal content text.
BigTextStyle bigStyle = new NotificationCompat.BigTextStyle();
bigStyle.bigText(eventDescription);

NotificationCompat.Builder notificationBuilder =
        new NotificationCompat.Builder(this)
        .setSmallIcon(R.drawable.ic_event)
        .setLargeIcon(BitmapFactory.decodeResource(
```

```
                getResources(), R.drawable.notif_background))
        .setContentTitle(eventTitle)
        .setContentText(eventLocation)
        .setContentIntent(viewPendingIntent)
        .addAction(R.drawable.ic_map,
                getString(R.string.map), mapPendingIntent)
        .setStyle(bigStyle);
```

Notice that you can add a large icon image to any notification using the `setLargeIcon()` method. However, these icons appear as large background images on a wearable and do not look good as they are scaled up to fit the wearable screen. To add a wearable-specific background image to a notification, see Add Wearable Features For a Notification. For more information about designing notifications with large images, see the Design Principles of Android Wear.

Add Wearable Features For a Notification

If you ever need to add wearable-specific options to a notification, such as specifying additional pages of content or letting users dictate a text response with voice input, you can use the `NotificationCompat.WearableExtender` class to specify the options. To use this API:

1. Create an instance of a `WearableExtender`, setting the wearable-specific options for the notication.
2. Create an instance of `NotificationCompat.Builder`, setting the desired properties for your notification as described earlier in this lesson.
3. Call `extend()` on the notification and pass in the `WearableExtender`. This applies the wearable options to the notification.
4. Call `build()` to build the notification.

For example, the following code calls the `setHintHideIcon()` method to remove the app icon from the notification card.

```
// Create a WearableExtender to add functionality for wearables
NotificationCompat.WearableExtender wearableExtender =
        new NotificationCompat.WearableExtender()
        .setHintHideIcon(true)
        .setBackground(mBitmap);

// Create a NotificationCompat.Builder to build a standard
```

```
notification
// then extend it with the WearableExtender
Notification notif = new NotificationCompat.Builder(mContext)
        .setContentTitle("New mail from " + sender)
        .setContentText(subject)
        .setSmallIcon(R.drawable.new_mail)
        .extend(wearableExtender)
        .build();
```

The `setHintHideIcon()` and `setBackground()` methods are just two examples of new notification features available with `NotificationCompat.WearableExtender`.

Note: The bitmap that you use with `setBackground()` should have a resolution of 400x400 for non-scrolling backgrounds and 640x400 for backgrounds that support parallax scrolling. Place these bitmap images in the `res/drawable-nodpi` directory. Place other non-bitmap resources for wearable notifications, such as those used with the `setContentIcon()` method, in the `res/drawable-hdpi` directory.

If you ever need to read wearable-specific options at a later time, use the corresponding get method for the option. This example calls the `getHintHideIcon()` method to get whether or not this notification hides the icon:

```
NotificationCompat.WearableExtender wearableExtender =
        new NotificationCompat.WearableExtender(notif);
boolean hintHideIcon = wearableExtender.getHintHideIcon();
```

Deliver the Notification

When you want to deliver your notifications, always use the `NotificationManagerCompat` API instead of `NotificationManager`:

```
// Get an instance of the NotificationManager service
NotificationManagerCompat notificationManager =
        NotificationManagerCompat.from(mContext);

// Issue the notification with notification manager.
notificationManager.notify(notificationId, notif);
```

If you use the framework's `NotificationManager`, some features from `NotificationCompat.WearableExtender` do not work, so make sure to use `NotificationCompat`.

Next Get started Next class

Receiving Voice Input in a Notification

Previous Next Get started

This lesson teaches you to

1. Define the Voice Input
2. Add the Voice Input as a Notification Action
3. Receiving the Voice Input as a String

If you have handheld notifications that include an action to input text, such as reply to an email, it should normally launch an activity on the handheld device to input the text. However, when your notification appears on a wearable, there is no keyboard input, so you can let users dictate a reply or provide pre-defined text messages using `RemoteInput`.

When users reply with voice or select one of the available messages, the system attaches the text response to the `Intent` you specified for the notification action and sends that intent to your handheld app.

Note: The Android emulator does not support voice input. When using the emulator for a wearable device, enable **Hardware keyboard present** in the AVD settings so you can type replies instead.

Define the Voice Input

To create an action that supports voice input, create an instance of `RemoteInput.Builder` that you can add to your notification action. This class's constructor accepts a string that the system uses as the key for the voice input, which you'll later use to retrieve the text of the input in your handheld app.

For example, here's how to create a `RemoteInput` object that provides a custom label for the voice input prompt:

```
// Key for the string that's delivered in the action's intent
private static final String EXTRA_VOICE_REPLY = "extra_voice_reply";

String replyLabel = getResources().getString(R.string.reply_label);

RemoteInput remoteInput = new RemoteInput.Builder(EXTRA_VOICE_REPLY)
        .setLabel(replyLabel)
        .build();
```

Add Pre-defined Text Responses

In addition to allowing voice input, you can provide up to five text responses that the user can select for quick replies. Call `setChoices()` and pass it a string array.

For example, you can define some responses in a resource array:

res/values/strings.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="reply_choices">
        <item>Yes</item>
        <item>No</item>
        <item>Maybe</item>
    </string-array>
</resources>
```

Then, inflate the string array and add it to the `RemoteInput`:

```
public static final String EXTRA_VOICE_REPLY = "extra_voice_reply";
...
String replyLabel = getResources().getString(R.string.reply_label);
String[] replyChoices =
getResources().getStringArray(R.array.reply_choices);

RemoteInput remoteInput = new RemoteInput.Builder(EXTRA_VOICE_REPLY)
        .setLabel(replyLabel)
```

```
        .setChoices(replyChoices)
        .build();
```

Add the Voice Input as a Notification Action

To set the voice input, attach your `RemoteInput` object to an action using `addRemoteInput()`. You can then apply the action to the notification. For example:

```
// Create an intent for the reply action
Intent replyIntent = new Intent(this, ReplyActivity.class);
PendingIntent replyPendingIntent =
        PendingIntent.getActivity(this, 0, replyIntent,
                PendingIntent.FLAG_UPDATE_CURRENT);

// Create the reply action and add the remote input
NotificationCompat.Action action =
        new
NotificationCompat.Action.Builder(R.drawable.ic_reply_icon,
                getString(R.string.label), replyPendingIntent)
                .addRemoteInput(remoteInput)
                .build();

// Build the notification and add the action via WearableExtender
Notification notification =
        new NotificationCompat.Builder(mContext)
                .setSmallIcon(R.drawable.ic_message)
                .setContentTitle(getString(R.string.title))
                .setContentText(getString(R.string.content))
                .extend(new WearableExtender().addAction(action))
                .build();

// Issue the notification
NotificationManagerCompat notificationManager =
        NotificationManagerCompat.from(mContext);
notificationManager.notify(notificationId, notification);
```

When you issue this notification, users can swipe to the left to see the "Reply" action button.

Receiving the Voice Input as a String

To receive the user's transcribed message in the activity you declared in the reply action's intent, call `getResultsFromIntent()`, passing in the "Reply" action's intent. This method returns a `Bundle` that contains the text response. You can then query the `Bundle` to obtain the response.

Note: Do not use `Intent.getExtras()` to obtain the voice result, because

the voice input is stored as `ClipData`. The `getResultsFromIntent()` method provides a convenient way to receive a character sequence without having to process the `ClipData` yourself.

The following code shows a method that accepts an intent and returns the voice response, which is referenced by the `EXTRA_VOICE_REPLY` key that is used in the previous examples:

```
/**
 * Obtain the intent that started this activity by calling
 * Activity.getIntent() and pass it into this method to
 * get the associated voice input string.
 */

private CharSequence getMessageText(Intent intent) {
    Bundle remoteInput = RemoteInput.getResultsFromIntent(intent);
    if (remoteInput != null) {
        return remoteInput.getCharSequence(EXTRA_VOICE_REPLY);
    }
    return null;
}
```

Next Get started Next class

Adding Pages to a Notification

Previous Next Get started

This lesson teaches you to

1. Add Pages to a Notification

When you'd like to provide more information without requiring users to open your app on their handheld device, you can add one or more pages to the notification on the wearable. The additional pages appear immediately to the right of the main notification card.

To create a notification with multiple pages:

1. Create the main notification (the first page) with `NotificationCompat.Builder`, in the way you'd like the notification to appear on a handset.
2. Create the additional pages for the wearable with `NotificationCompat.Builder`.
3. Apply the pages to the main notification with the `addPage()` method or add multiple pages in a `Collection` with the `addPages()` method.

For example, here's some code that adds a second page to a notification:

```
// Create builder for the main notification
NotificationCompat.Builder notificationBuilder =
        new NotificationCompat.Builder(this)
        .setSmallIcon(R.drawable.new_message)
        .setContentTitle("Page 1")
        .setContentText("Short message")
        .setContentIntent(viewPendingIntent);

// Create a big text style for the second page
BigTextStyle secondPageStyle = new NotificationCompat.BigTextStyle();
secondPageStyle.setBigContentTitle("Page 2")
               .bigText("A lot of text...");

// Create second page notification
Notification secondPageNotification =
        new NotificationCompat.Builder(this)
        .setStyle(secondPageStyle)
        .build();

// Extend the notification builder with the second page
Notification notification = notificationBuilder
        .extend(new NotificationCompat.WearableExtender()
                .addPage(secondPageNotification))
        .build();

// Issue the notification
notificationManager =
        NotificationManagerCompat.from(this);
notificationManager.notify(notificationId, notification);
```

Next Get started Next class

Stacking Notifications

Previous Next Get started

This lesson teaches you to

1. Add Each Notification to a Group
2. Add a Summary Notification

When creating notifications for a handheld device, you should always aggregate similar notifications into a single summary notification. For example, if your app creates notifications for received messages, you should not show more than one notification on a handheld device—when more than one is message is received, use a single notification to provide a summary such as "2 new messages."

However, a summary notification is less useful on an Android wearable because users are not able to read details from each message on the wearable (they must open your app on the handheld to view more information). So for the wearable device, you should group all the notifications together in a stack. The stack of notifications appears as a single card, which users can expand to view the details from each notification separately. The new `setGroup()` method makes this possible while allowing you to still provide only one summary notification on the handheld device.

Add Each Notification to a Group

To create a stack, call `setGroup()` for each notification you want in the stack and specify a group key. Then call `notify()` to send it to the wearable.

```
final static String GROUP_KEY_EMAILS = "group_key_emails";

// Build the notification, setting the group appropriately
Notification notif = new NotificationCompat.Builder(mContext)
         .setContentTitle("New mail from " + sender1)
         .setContentText(subject1)
         .setSmallIcon(R.drawable.new_mail)
         .setGroup(GROUP_KEY_EMAILS)
         .build();

// Issue the notification
NotificationManagerCompat notificationManager =
        NotificationManagerCompat.from(this);
notificationManager.notify(notificationId1, notif);
```

Later on, when you create another notification, specify the same group key. When you call `notify()`, this notification appears in the same stack as the previous notification, instead of as a new card:

```
Notification notif2 = new NotificationCompat.Builder(mContext)
         .setContentTitle("New mail from " + sender2)
         .setContentText(subject2)
         .setSmallIcon(R.drawable.new_mail)
         .setGroup(GROUP_KEY_EMAILS)
         .build();

notificationManager.notify(notificationId2, notif2);
```

By default, notifications appear in the order in which you added them, with the most recent notification visible at the top. You can order notifications in another fashion by calling `setSortKey()`.

Add a Summary Notification

It's important that you still provide a summary notification that appears on handheld devices. So in addition to adding each unique notification to the

same stack group, also add a summary notification and call `setGroupSummary()` on the summary notification.

This notification does not appear in your stack of notifications on the wearable, but it appears as the only notification on the handheld device.

```
Bitmap largeIcon = BitmapFactory.decodeResource(getResources(),
        R.drawable.ic_large_icon);

// Create an InboxStyle notification
Notification summaryNotification = new
NotificationCompat.Builder(mContext)
        .setContentTitle("2 new messages")
        .setSmallIcon(R.drawable.ic_small_icon)
        .setLargeIcon(largeIcon)
        .setStyle(new NotificationCompat.InboxStyle()
                .addLine("Alex Faaborg   Check this out")
                .addLine("Jeff Chang   Launch Party")
                .setBigContentTitle("2 new messages")
                .setSummaryText("johndoe@gmail.com"))
        .setGroup(GROUP_KEY_EMAILS)
        .setGroupSummary(true)
        .build();

notificationManager.notify(notificationId3, summaryNotification);
```

This notification uses `NotificationCompat.InboxStyle`, which gives you an easy way to create notifications for email or messaging apps. You can use this style, another one defined in `NotificationCompat`, or no style for the summary notification.

Tip: To style the text like in the example screenshot, see Styling with HTML markup and Styling with Spannables.

Summary notifications can also affect notifications on wearables without being displayed on them. When creating a summary notification, you can

use the `NotificationCompat.WearableExtender` class and call `setBackground()` or `addAction()` to set a background image or an action that applies to the entire stack on the wearable. For instance, to set the background for an entire stack of notifications:

```
Bitmap background = BitmapFactory.decodeResource(getResources(),
        R.drawable.ic_background);

NotificationCompat.WearableExtender wearableExtender =
        new NotificationCompat.WearableExtender()
        .setBackground(background);

// Create an InboxStyle notification
Notification summaryNotificationWithBackground =
        new NotificationCompat.Builder(mContext)
        .setContentTitle("2 new messages")
        ...
        .extend(wearableExtender)
        .setGroup(GROUP_KEY_EMAILS)
        .setGroupSummary(true)
        .build();
```

Next Get started Next class

Creating Wearable Apps

Previous Next Get started

Dependencies and Prerequisites

- An Android Wear device

Wearable apps run directly on the device, giving you access to hardware such as sensors and the GPU. They are fundamentally the same as apps built for other devices using the Android SDK, but differ greatly in design and usability and the amount of functionality provided. These are the main differences between handheld and wearable apps:

- Wearable apps are relatively small in size and functionality compared to handheld apps. They contain only what makes sense on the wearable, which is usually a small subset of the corresponding handheld app. In general, you should carry out operations on the handheld when possible and send the results to the wearable.
- Users don't download apps directly onto the wearable. Instead, you bundle the wearable app inside the handheld app. When users install the handheld app, the system automatically installs the wearable app. However, for development purposes, you can still install the wearable app directly to the wearable.

- Wearable apps can access much of the standard Android APIs, but don't support the following APIs:

    - `android.webkit`
    - `android.print`
    - `android.app.backup`
    - `android.appwidget`
    - `android.hardware.usb`

You can check if a wearable supports a feature by calling `hasSystemFeature()` before trying to use an API.

To conserve power on a wearable device, you can enable ambient mode for your Wear app. Devices transition from interactive to ambient mode when the user is idle on an activity or when the user covers the screen with their palm. Wearable apps that can transition into ambient mode are called *always-on* apps. The following describes the two modes of operation for always-on apps:

Interactive
: Use full color with fluid animation in this mode. The app is also responsive to input.

Ambient
: Render the screen with grayscale graphics and do not present any input cues in this mode. This display mode is only supported on devices running Android 5.1 or higher.

On devices running versions prior to Android 5.1 or for apps that do not support ambient mode, when a user is idle on an activity or when the user covers the screen with their palm on an activity, the Wear home screen is displayed instead of your activity in ambient mode. If you need to show persistent content on versions prior to Android 5.1, create a notification in the context stream instead.

Note: We recommend using Android Studio for Android Wear development as it provides project setup, library inclusion, and packaging conveniences that aren't available in ADT. The rest of this training assumes you're using Android Studio.

Lessons

Creating and Running a Wearable App
: Learn how to create an Android Studio project that contains both the wearable and handheld app modules and how to run the app on a device or emulator.

Creating Custom Layouts

Learn how to create and display custom layouts for notifications and activities.

Keeping Your App Visible

Learn how to enable ambient mode for your apps, so they remain visible while still saving battery power.

Adding Voice Capabilities

Learn how to launch an activity with voice actions and how to start the system speech recognizer app to obtain free-form voice input.

Packaging Wearable Apps

Learn how to package a wearable app inside a handheld app. This allows the system to install the wearable app automatically when users install the companion handheld app from the Google Play store.

Debugging over Bluetooth

Learn how to debug your wearable over Bluetooth instead of USB.

Next Get started Next class

Creating and Running a Wearable App

Previous Next Get started

This lesson teaches you to

1. Update Your SDK
2. Set Up an Android Wear Emulator or Device
3. Create a Project
4. Install the Wearable App
5. Include the Correct Libraries

Wearable apps run directly on the wearable device, giving you access to low-level hardware such as sensors, activities, services, and more, right on the wearable.

A companion handheld app that contains the wearable app is also required when you want to publish to the Google Play store. Wearables don't support the Google Play store, so users download the companion handheld app, which automatically pushes the wearable app to the wearable. The handheld app is also useful for doing heavy processing, network actions, or other work and sending the results to the wearable.

This lesson goes over how to set up a device or emulator and create one project to contain both your wearable and handheld apps.

Update Your SDK

Before you begin building wearable apps, you must:

- **Update your SDK tools to version 23.0.0 or higher**
  The updated SDK tools enable you to build and test wearable apps.

- **Update your SDK with Android 4.4W.2 (API 20) or higher**
  The updated platform version provides new APIs for wearable apps.

To update your SDK with these components, see Get the latest SDK tools.

Set Up an Android Wear Emulator or Device

We recommend that you develop on real hardware so you can better gauge the user experience. However, the emulator lets you test out different types of screen shapes, which is useful for testing.

Set up an Android Wear Virtual Device

To set up an Android Wear virtual device:

1. Click **Tools > Android > AVD Manager**.
2. Click **Create Virtual Device...**.
   1. Click **Wear** in the Category list:
   2. Select Android Wear Square or Android Wear Round.
   3. Click **Next**.
   4. Select a release name (for example, KitKat Wear).
   5. Click **Next**.
   6. (Optional) Change any preferences for your virtual device.
   7. Click **Finish**.
3. Start the emulator:
   1. Select the virtual device you just created.
   2. Click the **Play** button.
   3. Wait until the emulator initializes and shows the Android Wear home screen.
4. Pair your handheld with the emulator:
   1. On your handheld, install the Android Wear app from Google Play.
   2. Connect the handheld to your machine through USB.
   3. Forward the AVD's communication port to the connected handheld device (you must do this every time the handheld is connected):

```
adb -d forward tcp:5601 tcp:5601
```

4. Start the Android Wear app on your handheld device and connect to the emulator.
5. Tap the menu on the top right corner of the Android Wear app and select **Demo Cards**.
6. The cards you select appear as notifications on the home screen of the emulator.

Set Up an Android Wear Device

To set up an Android Wear device:

1. Install the Android Wear app, available on Google Play, on your handheld.
2. Follow the app's instructions to pair your handheld with your wearable. This allows you to test out synced handheld notifications, if you're building them.
3. Leave the Android Wear app open on your phone.
4. Enable adb debugging on the Android Wear device.
   1. Go to **Settings > About**.
   2. Tap **Build number** seven times.
   3. Swipe right to return to the Settings menu.
   4. Go to **Developer options** at the bottom of the screen.
   5. Tap **ADB Debugging** to enable adb.
5. Connect the wearable to your machine through USB, so you can install apps directly to it as you develop. A message appears on both the wearable and the Android Wear app prompting you to allow debugging.

   Note: If you can not connect your wearable to your machine via USB, you can try connecting over Bluetooth.

6. On the Android Wear app, check **Always allow from this computer** and tap **OK**.

The *Android* tool window on Android Studio shows the system log from the wearable. The wearable should also be listed when you run the `adb`

`devices` command.

Create a Project

To begin development, create an app project that contains wearable and handheld app modules. In Android Studio, click **File** > **New Project** and follow the Project Wizard instructions, as described in Creating a Project. As you follow the wizard, enter the following information:

1. In the **Configure your Project** window, enter a name for your app and a package name.
2. In the **Form Factors** window:
   - Select **Phone and Tablet** and select **API 9: Android 2.3 (Gingerbread)** under **Minimum SDK**.
   - Select **Wear** and select **API 20: Android 4.4 (KitKat Wear)** under **Minimum SDK**.
3. In the first **Add an Activity** window, add a blank activity for mobile.
4. In the second **Add an Activity** window, add a blank activity for Wear.

When the wizard completes, Android Studio creates a new project with two modules, **mobile** and **wear**. You now have a project for both your handheld and wearable apps for which you can create activities, services, and custom layouts. The handheld app does most of the heavy lifting, such as network communications, intensive processing, or tasks that require long amounts of user interaction. When the app completes these operations, your app should notify the wearable of the results through notifications or by syncing and sending data to the wearable.

Note: The **wear** module also contains a "Hello World" activity that uses a `WatchViewStub`. This class inflates a layout based on whether the device's screen is round or square. The `WatchViewStub` class is one of the UI widgets that the wearable support library provides.

Install the Wearable App

When developing, you install apps directly to the wearable like with

handheld apps. Use either `adb install` or the **Play** button on Android Studio.

When you're ready to publish your app to users, you embed the wearable app inside of the handheld app. When a user installs the handheld app from Google Play, a connected wearable automatically receives the wearable app.

Note: The automatic installation of wearable apps does not work when you are signing apps with a debug key and only works with release keys. See Packaging Wearable Apps for complete information on how to properly package wearable apps.

- To install the "Hello World" app to the wearable, select **wear** from the **Run/Debug configuration** drop-down menu and click the **Play** button. The activity shows up on the wearable and prints out "Hello world!"

Include the Correct Libraries

As part of the Project Wizard, the correct dependencies are imported for you in the appropriate module's `build.gradle` file. However, these dependencies are not required, so read the following descriptions to find out if you need them or not:

Notifications

The Android v4 support library (or v13, which includes v4) contains the APIs to extend your existing notifications on handhelds to support wearables.

For notifications that appear only on the wearable (meaning, they are issued by an app that runs on the wearable), you can just use the standard framework APIs (API Level 20) on the wearable and remove the support library dependency in the **mobile** module of your project.

Wearable Data Layer

To sync and send data between wearables and handhelds with the

Wearable Data Layer APIs, you need the latest version of Google Play services. If you're not using these APIs, remove the dependency from both modules.

Wearable UI support library

This is an unofficial library that includes UI widgets designed for wearables. We encourage you to use them in your apps, because they exemplify best practices, but they can still change at any time. However, if the libraries are updated, your apps won't break since they are compiled into your app. To get new features from an updated library, you just need to statically link the new version and update your app accordingly. This library is only applicable if you create wearable apps.

In the next lessons, you'll learn how to create layouts designed for wearables as well as how to use the various voice actions that are supported by the platform.

Next Get started Next class

Creating Custom Layouts

Previous Next Get started

This lesson teaches you to

1. Create custom notifications
2. Create Layouts with the Wearable UI Library

You should also read

- Android Wear Design Principles

Creating layouts for wearables is the same as handheld devices, except you have to design for the screen size and for glanceability. Do not port functionality and the UI from a handheld app and expect a good experience. You should create custom layouts only when necessary. Read the design guidelines for information on how to design great wearable apps.

Create Custom Notifications

In general, you should create notifications on the handheld and let them automatically sync to the wearable. This lets you build your notifications once and have them appear on many types of devices (not just wearables, but eventually Auto and TV) without having to design them for different form factors.

If the standard notification styles don't work for you (such as `NotificationCompat.BigTextStyle` or `NotificationCompat.InboxStyle`), you can display an activity with a custom layout. You can only create and issue custom notifications on the wearable, and the system does not sync these notifications to the handheld.

Note: When creating custom notifications on the wearable, you can use the standard notification APIs (API Level 20) instead of the Support Library.

To create a custom notification:

1. Create a layout and set it as the content view for the activity that you want to display.

   ```
   public void onCreate(Bundle bundle){
       ...
       setContentView(R.layout.notification_activity);
   }
   ```

2. Define necessary properties for the activity in the Android manifest to allow the activity to be displayed in the wearable's context stream process. You need to declare the activity to be exportable, be embeddable, and have an empty task affinity. We also recommend setting the theme to `Theme.DeviceDefault.Light`. For example:

   ```
   <activity android:name="com.example.MyDisplayActivity"
        android:exported="true"
        android:allowEmbedded="true"
        android:taskAffinity=""
        android:theme="@android:style/Theme.DeviceDefault.Light" />
   ```

3. Create a `PendingIntent` for the activity that you want to display. For example:

   ```
   Intent notificationIntent = new Intent(this,
   NotificationActivity.class);
   PendingIntent notificationPendingIntent =
   PendingIntent.getActivity(
           this, 0, notificationIntent,
   PendingIntent.FLAG_UPDATE_CURRENT);
   ```

4. Build a `Notification` and call `setDisplayIntent()` providing the `PendingIntent`. The system uses this `PendingIntent` to launch the activity when users view your notification.
5. Issue the notification using the `notify()` method.

   Note: When the notification is peeking on the homescreen, the system displays it with a standard template that it generates from the notification's semantic data. This template works well on all watchfaces. When users swipe the notification up, they'll then see

the custom activity for the notification.

Create Layouts with the Wearable UI Library

The Wearable UI Library is automatically included when you create your wearable app with the Android Studio Project Wizard. You can also add this library to your `build.gradle` file with the following dependency declaration:

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.google.android.support:wearable:+'
    compile 'com.google.android.gms:play-services-wearable:+'
}
```

This library helps you build UIs that are designed for wearables. For more information, see Creating Custom UIs for Wear Devices.

Here are some of the major classes in the Wearable UI Library:

`BoxInsetLayout`
: A `FrameLayout` object that's aware of screen shape and can box its children in the center square of a round screen.

`CardFragment`
: A fragment that presents content within an expandable, vertically scrollable card.

`CircledImageView`
: An image view surrounded by a circle.

`ConfirmationActivity`
: An activity that displays confirmation animations after the user completes an action.

`CrossFadeDrawable`
: A drawable that contains two child drawables and provides methods to directly adjust the blend between the two.

`DelayedConfirmationView`
: A view that provides a circular countdown timer, typically used to automatically confirm an operation after a short delay has elapsed.

`DismissOverlayView`

A view for implementing long-press-to-dismiss.

`GridViewPager`
: A layout manager that allows the user to navigate both vertically and horizontally through pages of data. You supply an implementation of a `GridPagerAdapter` instance to generate the pages that the view shows.

`GridPagerAdapter`
: An adapter that supplies pages to a `GridViewPager` object.

`FragmentGridPagerAdapter`
: An implementation of a `GridPagerAdapter` instance that represents each page as a fragment.

`DotsPageIndicator`
: A page indicator for a `GridViewPager` implementation that identifies the current page in relation to all available pages on the current row.

`WatchViewStub`
: A class that can inflate a specific layout, depending on the shape of the device's screen.

`WearableListView`
: An alternative version of a `ListView` object that is optimized for ease of use on small screen wearable devices. It displays a vertically scrollable list of items, and automatically snaps to the nearest item when the user stops scrolling.

Wear UI library API reference

The reference documentation explains how to use each UI widget in detail. Browse the Wear API reference documentation for the classes above.

Download the Wearable UI library for Eclipse ADT

If you are using the ADT plugin for Eclipse, download the Wearable UI library to include the Wearable UI library as a dependency in your project.

Note: We recommend Android Studio for Android Wear app development.

Next Get started Next class

Keeping Your App Visible

Previous Next Get started

This lesson teaches you to

1. Enable Ambient Mode in a Wearable App
2. Update Content in Ambient Mode
3. Maintain Backward-compatibility

You should also read

- App Structure for Android Wear

Related Samples

- AlwaysOn

Some Wear apps are most useful when they are constantly visible to the user. For example, users out on a run can glance at their wearable to see the distance covered and time elapsed, or after recording a grocery list on their wearable, users can quickly see which items are remaining on the list as they shop at the market. Making an app constantly visible has an impact on battery life, so you should carefully consider that impact when adding this feature to your app.

Android Wear devices running Android version 5.1 or higher allow apps to remain in the foreground while saving battery power. Android Wear apps can control what's displayed on the wearable device screen while the device is in a low-power ambient mode. Wear apps that run in both ambient and interactive mode are called *always-on* apps.

This lesson describes how to enable your wearable app to be always-on, update the screen while in ambient mode, and maintain backwards compatibility.

Enable Ambient Mode in a Wearable App

For new and existing projects, you can add ambient mode support to your Wear app by updating your development project configuration. After you complete the project configuration, extend the `WearableActivity` class, which provides all the methods you need to enable ambient mode in your app. The following sections describe these steps in detail.

Configure your development project

In order to support ambient mode in your Wear app, you must update your Android SDK and configure your development project. Follow these steps to make the necessary changes:

- Update your SDK to include the Android 5.1 (API 22) or higher platform, which provides the APIs to allow activities to go into ambient mode. For more information on how to update your SDK, see Adding SDK Packages.
- Create a project or modify an existing project to target Android 5.1 or higher. This means you must set the manifest `targetSdkVersion` to 22 or higher.
- Set the manifest `minSdkVersion` to 20 or higher, if you want to support devices on versions prior to Android 5.1. For more information on backwards compatibility, see Maintain Backward-compatibility.
- Add or update the following dependencies to your `build.gradle` file:

```
dependencies {
    ...
    compile 'com.google.android.support:wearable:1.2.0'
    provided 'com.google.android.wearable:wearable:1.0.0'
}
```

Note: The `provided` dependency ensures that the classes loaded at run-time to support ambient mode are also available at compile-time.

- Add the wearable shared library entry into the wearable app manifest:

```
<application>
  <uses-library android:name="com.google.android.wearable"
                android:required="false" />
  ...
</application>
```

- Add the `WAKE_LOCK` permission to the handheld and wearable app manifest:

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

Create an activity that supports ambient mode

To enable ambient mode in your activity, use the `WearableActivity` class and methods.

1. Create an activity that extends `WearableActivity`.
2. In the `onCreate()` method of your activity, call the `setAmbientEnabled()` method.

Enable ambient mode in your activity as follows:

```
public class MainActivity extends WearableActivity {

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setAmbientEnabled();
    ...
}
```

Handle transitions between modes

If the user does not interact with your app for a period of time while it is displayed, or if the user covers the screen with their palm, the system switches the activity to ambient mode. After the app switches to ambient mode, update the activity UI to a more basic layout to reduce power consumption. You should use a black background with minimal white graphics and text. To ease a user into the transition from interactive to ambient mode, try to maintain similar placement of items on the screen. For more information on presenting content on an ambient screen, see the Watch Faces for Android Wear design guide.

Note: In ambient mode, disable any interactive elements on the screen, such as buttons. For more information on how to design user interactions for an always-on app, see the App Structure for Android Wear design guide.

When the activity switches to ambient mode, the system calls the `onEnterAmbient()` method in your wearable activity. The following code snippet shows how to change the text color to white and disable anti-aliasing after the system switches to ambient mode:

```
@Override
public void onEnterAmbient(Bundle ambientDetails) {
    super.onEnterAmbient(ambientDetails);

    mStateTextView.setTextColor(Color.WHITE);
    mStateTextView.getPaint().setAntiAlias(false);
}
```

When the user taps the screen or brings up their wrist, the activity switches from ambient mode to interactive mode. The system calls the `onExitAmbient()` method. Override this method to update the UI layout so that your app displays in a full-color, interactive state.

The following code snippet shows how to change the text color to green and enable anti-aliasing when the system switches to interactive mode:

```
@Override
public void onExitAmbient() {
    super.onExitAmbient();

    mStateTextView.setTextColor(Color.GREEN);
    mStateTextView.getPaint().setAntiAlias(true);
}
```

Update Content in Ambient Mode

Ambient mode allows you to update the screen with new information for the user, but you must carefully balance display updates against the battery life. You should strongly consider only overriding the `onUpdateAmbient()` method to update the screen once a minute in ambient mode. If your app requires more frequent updates, take into consideration that there is a trade-off between battery life and the frequency of updates. To realize battery savings, updates should be no more than once every 10 seconds. In practice, however, you should update your app less frequently than that.

Update once a minute

In order to preserve battery power, most wear apps should not frequently update the screen while in ambient mode. We recommend designing your app to update the screen once per minute while in this mode. The system provides a callback method, `onUpdateAmbient()`, that allows you to update the screen at this recommended frequency.

To update your app content, override the `onUpdateAmbient()` method in your wearable activity:

```
@Override
public void onUpdateAmbient() {
    super.onUpdateAmbient();

    // Update the content
}
```

Update more frequently

For apps that require more frequent updates, such as a fitness, time-keeping, and travel information apps, use an `AlarmManager` object to wake the processor and update the screen more frequently.

To implement an alarm that updates content more frequently in ambient mode, follow these steps:

1. Prepare the alarm manager.

2. Set the frequency of the updates.
3. Schedule the next update when the activity switches to ambient mode or is currently in ambient mode.
4. Cancel the alarm when the activity switches to interactive mode or the activity is stopped

Note: The alarm manager may create new instances of your activity as they are triggered. To prevent this situation, ensure that your activity is declared with the `android:launchMode="singleInstance"` parameter in the manifest.

The following sections describe these steps in detail.

Prepare the alarm manager

The alarm manager launches a pending intent that updates the screen and schedules the next alarm. The following example shows how to declare the alarm manager and the pending intent in the `onCreate()` method of your activity:

```
private AlarmManager mAmbientStateAlarmManager;
private PendingIntent mAmbientStatePendingIntent;

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setAmbientEnabled();

    mAmbientStateAlarmManager =
        (AlarmManager) getSystemService(Context.ALARM_SERVICE);
    Intent ambientStateIntent =
        new Intent(getApplicationContext(), MainActivity.class);

    mAmbientStatePendingIntent = PendingIntent.getActivity(
        getApplicationContext(),
        0,
        ambientStateIntent,
        PendingIntent.FLAG_UPDATE_CURRENT);
    ...
}
```

When the alarm triggers and launches the pending intent, update the screen and schedule the next alarm by overriding the `onNewIntent()` method:

```
@Override
public void onNewIntent(Intent intent) {
    super.onNewIntent(intent);

    setIntent(intent);

    // Described in the following section
    refreshDisplayAndSetNextUpdate();
}
```

Update screen and schedule data updates

In this example activity, the alarm manager triggers every 20 seconds in ambient mode. When the timer ticks, the alarm triggers the intent to update the screen and then sets the delay for the next update.

The following example shows how to update information on the screen and set the alarm for the next update:

```
// Milliseconds between waking processor/screen for updates
private static final long AMBIENT_INTERVAL_MS =
TimeUnit.SECONDS.toMillis(20);

private void refreshDisplayAndSetNextUpdate() {

    if (isAmbient()) {
        // Implement data retrieval and update the screen for
ambient mode
    } else {
        // Implement data retrieval and update the screen for
interactive mode
    }

    long timeMs = System.currentTimeMillis();

    // Schedule a new alarm
    if (isAmbient()) {
        // Calculate the next trigger time
        long delayMs = AMBIENT_INTERVAL_MS - (timeMs %
AMBIENT_INTERVAL_MS);
        long triggerTimeMs = timeMs + delayMs;

        mAmbientStateAlarmManager.setExact(
            AlarmManager.RTC_WAKEUP,
            triggerTimeMs,
            mAmbientStatePendingIntent);

    } else {
        // Calculate the next trigger time for interactive mode
    }
}
```

Schedule the next alarm

Schedule the alarm to update the screen when the activity is entering ambient mode or when the activity is already in ambient mode by overriding the `onEnterAmbient()` method and the `onUpdateAmbient()` method:

```
@Override
public void onEnterAmbient(Bundle ambientDetails) {
    super.onEnterAmbient(ambientDetails);

    refreshDisplayAndSetNextUpdate();
}

@Override
public void onUpdateAmbient() {
    super.onUpdateAmbient();

    refreshDisplayAndSetNextUpdate();
}
```

Note: In this example, the `refreshDisplayAndSetNextUpdate()` method is called whenever the screen needs to be updated. For more examples of when to call this method, see the AlwaysOn sample.

Cancel the alarm

When the device switches to interactive mode, cancel the alarm in the `onExitAmbient()` method:

```
@Override
public void onExitAmbient() {
    super.onExitAmbient();

    mAmbientStateAlarmManager.cancel(mAmbientStatePendingIntent);
}
```

When the user exits or stops your activity, cancel the alarm in the `onDestroy()` method of your activity:

```
@Override
public void onDestroy() {
    mAmbientStateAlarmManager.cancel(mAmbientStatePendingIntent);
    super.onDestroy();
}
```

Maintain Backward-compatibility

Activities that support ambient mode automatically fall back to normal activities on Wear devices that are on Android versions prior to 5.1 (API level 22). No special app code is required to support devices on these versions of Android. When the device switches to ambient mode, the device returns to the home screen and exits your activity.

If your app should not be installed or updated on devices with Android versions prior to 5.1, update your manifest with the following:

```
<uses-library android:name="com.google.android.wearable"
android:required="true" />
```

Next Get started Next class

Adding Voice Capabilities

Previous Next Get started

This lesson teaches you to

1. Declare System-provided Voice Actions
2. Declare App-provided Voice Actions
3. Obtaining Free-form Speech Input

You should also read

- Android Wear Design Principles

Voice actions are an important part of the wearable experience. They let users carry out actions hands-free and quickly. Wear provides two types of voice actions:

System-provided
: These voice actions are task-based and are built into the Wear platform. You filter for them in the activity that you want to start when the voice action is spoken. Examples include "Take a note" or "Set an alarm".

App-provided
: These voice actions are app-based, and you declare them just like a launcher icon. Users say "Start " to use these voice actions and an activity that you specify starts.

Declare System-provided Voice Actions

The Android Wear platform provides several voice intents that are based on user actions such as "Take a note" or "Set an alarm". This allows users to say what they want to do and let the system figure out the best activity to start.

When users speak the voice action, your app can filter for the intent that is fired to start an activity. If you want to start a service to do something in the background, show an activity as a visual cue and start the service in the activity. Make sure to call `finish()` when you want to get rid of the visual cue.

For example, for the "Take a note" command, declare this intent filter to start an activity named `MyNoteActivity`:

```
<activity android:name="MyNoteActivity">
    <intent-filter>
        <action android:name="android.intent.action.SEND" />
        <category android:name="com.google.android.voicesearch.SELF_NOTE" />
    </intent-filter>
```

```
</activity>
```

Here is a list of the voice intents supported by the Wear platform:

Name	Example Phrases	Intent
Call a car/taxi	"OK Google, get me a taxi" "OK Google, call me a car"	Action `com.google.android.gms.actions.RESERVE_TAXI_RESERVATION`

Take a note	"OK Google, take a note"
	"OK Google, note to self"
		Action
		`android.intent.action.SEND`
		Category
		`com.google.android.voicesearch.SELF_NOTE`
		Extras
		`android.content.Intent.EXTRA_TEXT` - a string with note body

Set alarm	"OK Google, set an alarm for 8 AM" "OK Google, wake me up at 6 tomorrow"	Action `android.intent.action.SET_ALARM` Extras `android.provider.AlarmClock.EXTRA_HOUR` - an integer with the hour of the alarm. `android.provider.AlarmClock.EXTRA_MINUTES` - an integer with the minute of the alarm (these 2 extras are optional, either none or both are provided)
Set timer	"Ok Google, set a timer for 10 minutes"	Action `android.intent.action.SET_TIMER` Extras `android.provider.AlarmClock.EXTRA_LENGTH` - an integer in the range of 1 to 86400 (number of seconds in 24 hours) representing the length of the timer
Start stopwatch	"Ok Google, start	Action `com.google.android.wearable.action.STOPWATCH`

	stopwatch"	

Start/Stop a bike ride	"OK Google, start cycling"
	"OK Google, start my bike ride"
	"OK Google, stop cycling"
		Action
		`vnd.google.fitness.TRACK`
		Mime Type
		`vnd.google.fitness.activity/biking`
		Extras
		`actionStatus` - a string with the value `ActiveActionStatus` when starting and `CompletedActionStatus` when stopping.

Start/Stop a run	"OK Google, track my run"
	"OK Google, start running"
	"OK Google, stop running"
		Action
		`vnd.google.fitness.TRACK`
		MimeType
		`vnd.google.fitness.activity/running`
		Extras
		`actionStatus` - a string with the value `ActiveActionStatus` when starting and `CompletedActionStatus` when stopping

Start/Stop a workout	"OK Google, start a workout"
	"OK Google, track my workout"
	"OK Google, stop workout"
		Action
		`vnd.google.fitness.TRACK`
		MimeType
		`vnd.google.fitness.activity/other`
		Extras
		`actionStatus` - a string with the value `ActiveActionStatus` when starting and `CompletedActionStatus` when stopping

	"OK Google,	

Show heart rate	what's my heart rate? "
	"OK Google, what's my bpm?"
		Action
		`vnd.google.fitness.VIEW`
		Mime Type
		`vnd.google.fitness.data_type/com.google.heart_rate.bpm`

Show step count	"OK Google, how many steps have I taken?"
	"OK Google, what's my step count?"
		Action
		`vnd.google.fitness.VIEW`
		Mime Type
		`vnd.google.fitness.data_type/com.google.step_count.cumulative`

For documentation on registering for platform intents and accessing the extras information contained in them, see Common intents.

Declare App-provided Voice Actions

If none of the platform voice intents work for you, you can start your apps directly with a "Start MyActivityName" voice action.

Registering for a "Start" action is the same as registering for a launcher icon on a handheld. Instead of requesting an app icon in a launcher, your app requests a voice action instead.

To specify the text to say after "Start", specify a `label` attribute for the activtiy that you want to start. For example, this intent filter recognizes the "Start MyRunningApp" voice action and launches `StartRunActivity`.

```
<application>
  <activity android:name="StartRunActivity" android:label="MyRunningApp">
      <intent-filter>
          <action android:name="android.intent.action.MAIN" />
          <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
  </activity>
</application>
```

Obtaining Free-form Speech Input

In addition to using voice actions to launch activities, you can also call the system's built-in

Speech Recognizer activity to obtain speech input from users. This is useful to obtain input from users and then process it, such as doing a search or sending it as a message.

In your app, you call `startActivityForResult()` using the `ACTION_RECOGNIZE_SPEECH` action. This starts the speech recognition activity, and you can then handle the result in `onActivityResult()`.

```
private static final int SPEECH_REQUEST_CODE = 0;

// Create an intent that can start the Speech Recognizer activity
private void displaySpeechRecognizer() {
    Intent intent = new Intent(RecognizerIntent.ACTION_RECOGNIZE_SPEECH);
    intent.putExtra(RecognizerIntent.EXTRA_LANGUAGE_MODEL,
            RecognizerIntent.LANGUAGE_MODEL_FREE_FORM);
// Start the activity, the intent will be populated with the speech text
    startActivityForResult(intent, SPEECH_REQUEST_CODE);
}

// This callback is invoked when the Speech Recognizer returns.
// This is where you process the intent and extract the speech text from the intent.
@Override
protected void onActivityResult(int requestCode, int resultCode,
        Intent data) {
    if (requestCode == SPEECH_REQUEST_CODE && resultCode == RESULT_OK) {
        List<String> results = data.getStringArrayListExtra(
                RecognizerIntent.EXTRA_RESULTS);
        String spokenText = results.get(0);
        // Do something with spokenText
    }
    super.onActivityResult(requestCode, resultCode, data);
}
```

Next Get started Next class

Packaging Wearable Apps

Previous Next Get started

This lesson teaches you to

1. Package with Android Studio
2. Package Manually
3. Turn off Asset Compression

When publishing to users, you must package a wearable app inside of a handheld app, because users cannot browse and install apps directly on the wearable. If packaged properly, when users download the handheld app, the system automatically pushes the wearable app to the paired wearable.

Note: This feature doesn't work when you are signing your apps with a debug key when developing. While developing, installing apps with `adb install` or Android Studio directly to the wearable is required.

Package with Android Studio

To properly package a wearable app in Android Studio:

1. Include all the permissions declared in the manifest file of the wearable app module in the manifest file of the handheld app module. For example, if you specify the `VIBRATE` permission for the wearable app, you must also add that permission to the handheld app.
2. Ensure that both the wearable and handheld app modules have the same package name and version number.
3. Declare a Gradle dependency in the handheld app's `build.gradle` file that points to the wearable app module:

```
dependencies {
   compile 'com.google.android.gms:play-services:5.0.+@aar'
   compile 'com.android.support:support-v4:20.0.+''
   wearApp project(':wearable')
}
```

4. Click **Build > Generate Signed APK...** and follow the on-screen instructions to specify your release keystore and sign your app. Android Studio exports the signed handheld app with the wearable app embedded in it automatically into your project's root folder.

   Alternatively, you can sign both apps from the command line using the Gradle wrapper. Both apps must be signed to have the automatic pushing of the wearable app work.

   Store your key file location and credentials in environment variables and run the Gradle wrapper as follows:

```
./gradlew assembleRelease \
  -Pandroid.injected.signing.store.file=$KEYFILE \
  -Pandroid.injected.signing.store.password=$STORE_PASSWORD \
  -Pandroid.injected.signing.key.alias=$KEY_ALIAS \
  -Pandroid.injected.signing.key.password=$KEY_PASSWORD
```

Signing the wearable and handheld app separately

If your build process requires signing the wearable app separately from the handheld app, you can declare the following Gradle rule in the handheld module's `build.gradle` to embed the previously-signed wearable app:

```
dependencies {
  ...
  wearApp files('/path/to/wearable_app.apk')
}
```

You then sign your handheld app in any manner you wish (either with the Android Studio **Build > Generate Signed APK...** menu item or with Gradle `signingConfig` rules as described in the previous section.

Package Manually

It's still possible to package the wearable app into the handheld app

manually if you are using another IDE or another method of building.

1. Include all the permissions declared in the manifest file of the wearable app in the manifest file of the mobile app. For example, if you specify the `VIBRATE` permission for the wearable app, you must also add that permission to the mobile app.
2. Ensure that both the wearable and mobile APKs have the same package name and version number.
3. Copy the signed wearable app to your handheld project's `res/raw` directory. We'll refer to the APK as `wearable_app.apk`.
4. Create a `res/xml/wearable_app_desc.xml` file that contains the version and path information of the wearable app. For example:

   ```
   <wearableApp package="wearable.app.package.name">
     <versionCode>1</versionCode>
     <versionName>1.0</versionName>
     <rawPathResId>wearable_app</rawPathResId>
   </wearableApp>
   ```

   The `package`, `versionCode`, and `versionName` are the same values specified in the wearable app's `AndroidManifest.xml` file. The `rawPathResId` is the static variable name of the APK resource. For example, for `wearable_app.apk`, the static variable name is `wearable_app`.

5. Add a `meta-data` tag to your handheld app's `<application>` tag to reference the `wearable_app_desc.xml` file.

   ```
   <meta-data android:name="com.google.android.wearable.beta.app"
                  android:resource="@xml/wearable_app_desc"/>
   ```

6. Build and sign the handheld app.

Turn off Asset Compression

Many build tools automatically compress any files added to the `res/raw` directory of an Android app. Because the wearable APK is already zipped, these tools re-compress the wearable APK and the wearable app installer can no longer read the wearable app.

When this happens, the installation fails. On the handheld app, the

`PackageUpdateService` logs the following error: "this file cannot be opened as a file descriptor; it is probably compressed."

Android Studio doesn't compress your APK by default, but if you are using another build process, ensure that you don't doubly compress the wearable app.

Next Get started Next class

Debugging over Bluetooth

Previous Next Get started

This lesson teaches you to

1. Set Up Devices for Debugging
2. Set Up a Debugging Session
3. Debug Your App

You should also read

- Android Wear Design Principles

You can debug your wearable over Bluetooth by routing its debug output to the handheld device that's connected to your development machine.

Setup Devices for Debugging

1. Enable USB debugging on the handheld:
    - Open the Settings app and scroll to the bottom.
    - If it doesn't have a Developer Options setting, tap **About Phone** (or **About Tablet**), scroll to the bottom, and tap the build number 7 times.
    - Go back and tap **Developer Options**.
    - Enable **USB debugging**.
2. Enable Bluetooth debugging on the wearable:
    1. Tap the home screen twice to bring up the Wear menu.
    2. Scroll to the bottom and tap **Settings**.
    3. Scroll to the bottom. If there's no **Developer Options** item, tap **About**, and then tap the build number 7 times.
    4. Tap the **Developer Options** item.
    5. Enable **Debug over Bluetooth**.

Set Up a Debugging Session

1. On the handheld, open the Android Wear companion app.
2. Tap the menu on the top right and select **Settings**.
3. Enable **Debugging over Bluetooth**. You should see a tiny status summary appear under the option:

   ```
   Host: disconnected
   Target: connected
   ```

4. Connect the handheld to your machine over USB and run:

   ```
   adb forward tcp:4444 localabstract:/adb-hub
   adb connect localhost:4444
   ```

   Note: You can use any available port that you have access to.

In the Android Wear companion app, you should see the status change to:

```
Host: connected
Target: connected
```

Debug Your App

Your wearable should show up as `localhost:4444` when running `adb devices`. To run any `adb` command, use this format:

```
adb -s localhost:4444 <command>
```

If there are no other devices connected over TCP/IP (namely emulators), you can shorten the command to:

```
adb -e <command>
```

For example:

```
adb -e logcat
adb -e shell
adb -e bugreport
```

Next Get started Next class

Creating Custom UIs for Wear Devices

Previous Next Get started

Dependencies and Prerequisites

- Android 4.4W (API level 20) or higher on the wearable device

User interfaces for wearable apps differ significantly from those built for handheld devices. Apps for wearables should follow the Android Wear design principles and implement the recommended UI patterns, which ensure a consistent user experience across apps that is optimized for wearables.

This class teaches you how to create custom UIs for your wearable apps and custom notifications that look good on any Android Wear device by implementing these UI patterns:

- Cards
- Countdowns and confirmations
- Long press to dismiss
- 2D Pickers
- Selection lists

The Wearable UI Library, which is part of the Google Repository in the Android SDK, provides classes that help you implement these patterns and create layouts that work on both round and square Android Wear devices.

Note: We recommend using Android Studio for Android Wear development as it provides project setup, library inclusion, and packaging conveniences that aren't available in ADT. This training assumes you are

using Android Studio.

Lessons

Defining Layouts
: Learn how to create layouts that look good on round and square Android Wear devices.

Creating Cards
: Learn how to create cards with custom layouts.

Creating Lists
: Learn how to create lists that are optimized for wearable devices.

Creating a 2D Picker
: Learn how to implement the 2D Picker UI pattern to navigate through pages of data.

Showing Confirmations
: Learn how to display confirmation animations when users complete actions.

Exiting Full-Screen Activities
: Learn how to implement the long-press-to-dismiss UI pattern to exit full-screen activities.

Next Get started Next class

Defining Layouts

Previous Next Get started

This lesson teaches you to

1. Add the Wearable UI Library
2. Specify Different Layouts for Square and Round Screens
3. Use a Shape-Aware Layout

You should also read

- Android Wear Design Principles

Video

- Full Screen Apps for Android Wear

Wearables use the same layout techniques as handheld Android devices, but need to be designed with specific constraints. Do not port functionality and the UI from a handheld app and expect a good experience. For more information on how to design great wearable apps, read the Android Wear Design Guidelines.

When you create layouts for Android Wear apps, you need to account for devices with square and round screens. Any content placed near the corners of the screen may be cropped on round Android Wear devices, so layouts designed for square screens do not work well on round devices. For a demonstration of this type of problem, see the video Full Screen Apps for Android Wear.

For example, figure 1 shows how the following layout looks on square and round screens:

Figure 1. Demonstration of how a layout designed for square screens does not work well on round screens.

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <TextView
        android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_square" />
</LinearLayout>
```

The text does not display correctly on devices with round screens.

The Wearable UI Library provides two different approaches to solve this problem:

- Define different layouts for square and round devices. Your app detects the shape of the device screen and inflates the correct layout at runtime.
- Use a special layout included in the library for both square and round devices. This layout applies different window insets depending on the shape of the device screen.

You typically use the first approach when you want your app to look

different depending on the shape of the device screen. You use the second approach when you want to use a similar layout on both screen shapes without having views cropped near the edges of round screens.

Add the Wearable UI Library

Android Studio includes the Wearable UI Library on your `wear` module by default when you use the Project Wizard. To compile your project with this library, ensure that the *Extras* > *Google Repository* package is installed in the Android SDK manager and that the following dependency is included in the `build.gradle` file of your `wear` module:

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.google.android.support:wearable:+'
    compile 'com.google.android.gms:play-services-wearable:+'
}
```

The `'com.google.android.support:wearable'` dependency is required to implement the layout techniques shown in the following sections.

Browse the API reference documentation for the Wearable UI Library classes.

Specify Different Layouts for Square and Round Screens

The `WatchViewStub` class included in the Wearable UI Library lets you specify different layout definitions for square and round screens. This class detects the screen shape at runtime and inflates the corresponding layout.

To use this class for handling different screen shapes in your app:

1. Add `WatchViewStub` as the main element of your activity's layout.
2. Specify a layout definition file for square screens with the `rectLayout` attribute.
3. Specify a layout definition file for round screens with the `roundLayout` attribute.

Define your activity's layout as follows:

```
<android.support.wearable.view.WatchViewStub
    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:id="@+id/watch_view_stub"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:rectLayout="@layout/rect_activity_wear"
    app:roundLayout="@layout/round_activity_wear">
</android.support.wearable.view.WatchViewStub>
```

Inflate this layout in your activity:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_wear);
}
```

Then create different layout definition files for square and round screens. In this example, you need to create the files `res/layout/rect_activity_wear.xml` and `res/layout/round_activity_wear.xml`. You define these layouts in the same way that you create layouts for handheld apps, but taking into account the constraints of wearable devices. The system inflates the correct layout at runtime depending on the screen shape.

Accessing layout views

The layouts that you specify for square or round screens are not inflated until `WatchViewStub` detects the shape of the screen, so your app cannot access their views immediately. To access these views, set a listener in your activity to be notified when the shape-specific layout has been inflated:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_wear);

    WatchViewStub stub = (WatchViewStub)
findViewById(R.id.watch_view_stub);
    stub.setOnLayoutInflatedListener(new
WatchViewStub.OnLayoutInflatedListener() {
        @Override public void onLayoutInflated(WatchViewStub stub) {
```

```
            // Now you can access your views
            TextView tv = (TextView) stub.findViewById(R.id.text);
            ...
        }
    });
}
```

Use a Shape-Aware Layout

The `BoxInsetLayout` class included in the Wearable UI Library extends `FrameLayout` and lets you define a single layout that works for both square and round screens. This class applies the required window insets depending on the screen shape and lets you easily align views on the center or near the edges of the screen.

Figure 2. Window insets on a round screen.

The gray square in figure 2 shows the area where `BoxInsetLayout` can automatically place its child views on round screens after applying the required window insets. To be displayed inside this area, children views specify the `layout_box` atribute with these values:

- A combination of `top`, `bottom`, `left`, and `right`. For example, `"left|top"` positions the child's left and top edges inside the gray square in figure 2.
- The `all` value positions all the child's content inside the gray square in figure 2.

On square screens, the window insets are zero and the `layout_box` attribute is ignored.

Figure 3. A layout definition that works on both square and round screens.

The layout shown in figure 3 uses the `<BoxInsetLayout>` element and works on square and round screens:

```
<android.support.wearable.view.BoxInsetLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:background="@drawable/robot_background"
    android:layout_height="match_parent"
    android:layout_width="match_parent"
    android:padding="15dp">

    <FrameLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:padding="5dp"
        app:layout_box="all">

        <TextView
            android:gravity="center"
            android:layout_height="wrap_content"
            android:layout_width="match_parent"
            android:text="@string/sometext"
            android:textColor="@color/black" />

        <ImageButton
            android:background="@null"
            android:layout_gravity="bottom|left"
            android:layout_height="50dp"
            android:layout_width="50dp"
            android:src="@drawable/ok" />

        <ImageButton
            android:background="@null"
            android:layout_gravity="bottom|right"
            android:layout_height="50dp"
```

```
            android:layout_width="50dp"
            android:src="@drawable/cancel" />
    </FrameLayout>
</android.support.wearable.view.BoxInsetLayout>
```

Notice the parts of the layout marked in bold:

- `android:padding="15dp"`

  This line assigns padding to the `<BoxInsetLayout>` element. Because the window insets on round devices are larger than 15dp, this padding only applies to square screens.

- `android:padding="5dp"`

  This line assigns padding to the inner `FrameLayout` element. This padding applies to both square and round screens. The total padding between the buttons and the window insets is 20 dp on square screens (15+5) and 5 dp on round screens.

- `app:layout_box="all"`

  This line ensures that the `FrameLayout` element and its children are boxed inside the area defined by the window insets on round screens. This line has no effect on square screens.

Next Get started Next class

Creating Cards

Previous Next Get started

This lesson teaches you to

1. Create a Card Fragment
2. Add a Card to Your Layout

You should also read

- Android Wear Design Principles

Cards present information to users with a consistent look and feel across different apps. This lesson shows you how to create cards in your Android Wear apps.

The Wearable UI Library provides implementations of cards specifically designed for wearable devices. This library contains the `CardFrame` class, which wraps views inside a card-styled frame with a white background, rounded corners, and a light-drop shadow. A `CardFrame` instance can only contain one direct child, usually a layout manager, to which you can add other views to customize the content inside the card.

You can add cards to your app in two ways:

- Use or extend the `CardFragment` class.
- Add a card inside a `CardScrollView` instance in your layout.

Note: This lesson shows you how to add cards to Android Wear activities. Android notifications on wearable devices are also displayed as cards. For more information, see Adding Wearable Features to Notifications.

Create a Card Fragment

The `CardFragment` class provides a default card layout with a title, a description, and an icon. Use this approach to add cards to your app if the default card layout shown in figure 1 meets your needs.

Figure 1. The default `CardFragment` layout.

To add a `CardFragment` instance to your app:

1. In your layout, assign an ID to the element that contains the card
2. Create a `CardFragment` instance in your activity
3. Use the fragment manager to add the `CardFragment` instance to its container

The following sample code shows the code for the screen display shown in figure 1:

```
<android.support.wearable.view.BoxInsetLayout
xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res-auto"
android:background="@drawable/robot_background"
android:layout_height="match_parent"
android:layout_width="match_parent">

    <FrameLayout
        android:id="@+id/frame_layout"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:layout_box="bottom">
```

```
    </FrameLayout>
</android.support.wearable.view.BoxInsetLayout>
```

The following code adds the `CardFragment` instance to the activity in figure 1:

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_wear_activity2);

    FragmentManager fragmentManager = getFragmentManager();
    FragmentTransaction fragmentTransaction =
fragmentManager.beginTransaction();
    CardFragment cardFragment =
CardFragment.create(getString(R.string.cftitle),

getString(R.string.cfdesc),
                                                    R.drawable.p);
    fragmentTransaction.add(R.id.frame_layout, cardFragment);
    fragmentTransaction.commit();
}
```

To create a card with a custom layout using the `CardFragment` class, extend the class and override its `onCreateContentView` method.

Add a CardFrame to Your Layout

You can also add a card directly to your layout definition, as shown in figure 2. Use this approach when you want to define a custom layout for the card inside a layout definition file.

Figure 2. Adding a `CardFrame` to your layout.

The following layout code sample demonstrates a vertical linear layout with two elements. You can create more complex layouts to fit the needs of your app.

```
<android.support.wearable.view.BoxInsetLayout
xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res-auto"
android:background="@drawable/robot_background"
android:layout_height="match_parent"
android:layout_width="match_parent">

    <android.support.wearable.view.CardScrollView
        android:id="@+id/card_scroll_view"
        android:layout_height="match_parent"
        android:layout_width="match_parent"
        app:layout_box="bottom">

        <android.support.wearable.view.CardFrame
            android:layout_height="wrap_content"
            android:layout_width="fill_parent">

            <LinearLayout
                android:layout_height="wrap_content"
                android:layout_width="match_parent"
                android:orientation="vertical"
                android:paddingLeft="5dp">
                <TextView
                    android:fontFamily="sans-serif-light"
                    android:layout_height="wrap_content"
                    android:layout_width="match_parent"
                    android:text="@string/custom_card"
                    android:textColor="@color/black"
                    android:textSize="20sp"/>
                <TextView
                    android:fontFamily="sans-serif-light"
                    android:layout_height="wrap_content"
                    android:layout_width="match_parent"
                    android:text="@string/description"
                    android:textColor="@color/black"
                    android:textSize="14sp"/>
            </LinearLayout>
        </android.support.wearable.view.CardFrame>
    </android.support.wearable.view.CardScrollView>
</android.support.wearable.view.BoxInsetLayout>
```

The `<CardScrollView>` element detects the shape of the screen and displays the card differently on round and square devices, using wider side margins on round screens. However, placing the `<CardScrollView>` element inside a `<BoxInsetLayout>` and using the `layout_box="bottom"` attribute is useful to align the card to the bottom of round screens without cropping its content.

Next Get started Next class

Creating Lists

Previous Next Get started

This lesson teaches you to

1. Add a List View
2. Create a Layout Implementation for List Items
3. Create a Layout Definition for Items
4. Create an Adapter to Populate the List
5. Associate the Adapter and Set a Click Listener

Related Samples

- Notifications

You should also read

- Android Wear Design Principles

Lists let users select an item from a set of choices easily on wearable devices. This lesson shows you how to create lists in your Android Wear apps.

The Wearable UI Library includes the `WearableListView` class, which is a list implementation optimized for wearable devices.

To create a list in your Android Wear apps:

1. Add a `WearableListView` element to your activity's layout definition.
2. Create a custom layout implementation for your list items.
3. Use this implementation to create a layout definition file for your list items.

4. Create an adapter to populate the list.
5. Assign the adapter to the `WearableListView` element.

Figure 1. A list view on Android Wear.

These steps are described in detail in the following sections.

Add a List View

The following layout adds a list view to an activity using a `<BoxInsetLayout>` element, so the list is displayed properly on both round and square devices:

```
<android.support.wearable.view.BoxInsetLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:background="@drawable/robot_background"
    android:layout_height="match_parent"
    android:layout_width="match_parent">

    <FrameLayout
        android:id="@+id/frame_layout"
        android:layout_height="match_parent"
        android:layout_width="match_parent"
        app:layout_box="left|bottom|right">

        <android.support.wearable.view.WearableListView
            android:id="@+id/wearable_list"
            android:layout_height="match_parent"
            android:layout_width="match_parent">
        </android.support.wearable.view.WearableListView>
    </FrameLayout>
</android.support.wearable.view.BoxInsetLayout>
```

Create a Layout Implementation for List

Items

In many cases, each list item consists of an icon and a description. The Notifications sample implements a custom layout that extends LinearLayout to incorporate these two elements inside each list item. This layout also implements the methods in the WearableListView.OnCenterProximityListener interface to change the color of the item's icon and fade the text in response to events from the WearableListView element as the user scrolls through the list.

```
public class WearableListItemLayout extends LinearLayout
             implements WearableListView.OnCenterProximityListener {

    private ImageView mCircle;
    private TextView mName;

    private final float mFadedTextAlpha;
    private final int mFadedCircleColor;
    private final int mChosenCircleColor;

    public WearableListItemLayout(Context context) {
        this(context, null);
    }

    public WearableListItemLayout(Context context, AttributeSet
attrs) {
        this(context, attrs, 0);
    }

    public WearableListItemLayout(Context context, AttributeSet
attrs,
                                  int defStyle) {
        super(context, attrs, defStyle);

        mFadedTextAlpha = getResources()

.getInteger(R.integer.action_text_faded_alpha) / 100f;
        mFadedCircleColor = getResources().getColor(R.color.grey);
        mChosenCircleColor = getResources().getColor(R.color.blue);
    }

    // Get references to the icon and text in the item layout
definition
    @Override
    protected void onFinishInflate() {
        super.onFinishInflate();
        // These are defined in the layout file for list items
        // (see next section)
        mCircle = (ImageView) findViewById(R.id.circle);
        mName = (TextView) findViewById(R.id.name);
    }

    @Override
    public void onCenterPosition(boolean animate) {
```

```
        mName.setAlpha(1f);
        ((GradientDrawable)
mCircle.getDrawable()).setColor(mChosenCircleColor);
    }

    @Override
    public void onNonCenterPosition(boolean animate) {
        ((GradientDrawable)
mCircle.getDrawable()).setColor(mFadedCircleColor);
        mName.setAlpha(mFadedTextAlpha);
    }
}
```

You can also create animator objects to enlarge the icon of the center item in the list. You can use the `onCenterPosition()` and `onNonCenterPosition()` callback methods in the `WearableListView.OnCenterProximityListener` interface to manage your animators. For more information about animators, see Animating with ObjectAnimator.

Create a Layout Definition for Items

After you implement a custom layout for list items, you provide a layout definition file that specifies the layout parameters of each of the components inside a list item. The following layout definition uses the custom layout implementation from the previous section and defines an icon and a text view whose IDs match those in the layout implementation class:

res/layout/list_item.xml

```
<com.example.android.support.wearable.notifications.WearableListItemLay

    xmlns:android="http://schemas.android.com/apk/res/android"
    android:gravity="center_vertical"
    android:layout_width="match_parent"
    android:layout_height="80dp">
    <ImageView
        android:id="@+id/circle"
        android:layout_height="20dp"
        android:layout_margin="16dp"
        android:layout_width="20dp"
        android:src="@drawable/wl_circle"/>
    <TextView
        android:id="@+id/name"
        android:gravity="center_vertical|left"
        android:layout_width="wrap_content"
        android:layout_marginRight="16dp"
        android:layout_height="match_parent"
```

```
        android:fontFamily="sans-serif-condensed-light"
        android:lineSpacingExtra="-4sp"
        android:textColor="@color/text_color"
        android:textSize="16sp"/>
</com.example.android.support.wearable.notifications.WearableListItemLa
```

Create an Adapter to Populate the List

The adapter populates the `WearableListView.OnCenterProximityListener` element with content. The following simple adapter populates the list with elements based on an array of strings:

```
private static final class Adapter extends WearableListView.Adapter
{
    private String[] mDataset;
    private final Context mContext;
    private final LayoutInflater mInflater;

    // Provide a suitable constructor (depends on the kind of
dataset)
    public Adapter(Context context, String[] dataset) {
        mContext = context;
        mInflater = LayoutInflater.from(context);
        mDataset = dataset;
    }

    // Provide a reference to the type of views you're using
    public static class ItemViewHolder extends
WearableListView.ViewHolder {
        private TextView textView;
        public ItemViewHolder(View itemView) {
            super(itemView);
            // find the text view within the custom item's layout
            textView = (TextView) itemView.findViewById(R.id.name);
        }
    }

    // Create new views for list items
    // (invoked by the WearableListView's layout manager)
    @Override
    public WearableListView.ViewHolder onCreateViewHolder(ViewGroup
parent,
                                                          int
viewType) {
        // Inflate our custom layout for list items
        return new
ItemViewHolder(mInflater.inflate(R.layout.list_item, null));
    }

    // Replace the contents of a list item
    // Instead of creating new views, the list tries to recycle
existing ones
    // (invoked by the WearableListView's layout manager)
    @Override
    public void onBindViewHolder(WearableListView.ViewHolder holder,
```

```
                                int position) {
        // retrieve the text view
        ItemViewHolder itemHolder = (ItemViewHolder) holder;
        TextView view = itemHolder.textView;
        // replace text contents
        view.setText(mDataset[position]);
        // replace list item's metadata
        holder.itemView.setTag(position);
    }

    // Return the size of your dataset
    // (invoked by the WearableListView's layout manager)
    @Override
    public int getItemCount() {
        return mDataset.length;
    }
}
```

Associate the Adapter and Set a Click Listener

In your activity, obtain a reference to the WearableListView.OnCenterProximityListener element from your layout, assign an instance of the adapter to populate the list, and set a click listener to complete an action when the user selects a particular list item.

```
public class WearActivity extends Activity
                           implements WearableListView.ClickListener {

    // Sample dataset for the list
    String[] elements = { "List Item 1", "List Item 2", ... };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.my_list_activity);

        // Get the list component from the layout of the activity
        WearableListView listView =
            (WearableListView) findViewById(R.id.wearable_list);

        // Assign an adapter to the list
        listView.setAdapter(new Adapter(this, elements));

        // Set a click listener
        listView.setClickListener(this);
    }

    // WearableListView click listener
    @Override
    public void onClick(WearableListView.ViewHolder v) {
        Integer tag = (Integer) v.itemView.getTag();
        // use this data to complete some action ...
    }
```

```
    @Override
    public void onTopEmptyRegionClick() {
    }
}
```

Next Get started Next class

Creating a 2D Picker

Previous Next Get started

This lesson teaches you to

1. Add a Page Grid
2. Implement a Page Adapter

Related Samples

- GridViewPager

You should also read

- Android Wear Design Principles

The 2D Picker pattern in Android Wear allows users to navigate and choose from a set of items shown as pages. The Wearable UI Library lets you easily implement this pattern using a page grid, which is a layout manager that allows users to scroll vertically and horizontally through pages of data.

To implement this pattern, you add a `GridViewPager` element to the layout of your activity and implement an adapter that provides a set of pages by extending the `FragmentGridPagerAdapter` class.

Add a Page Grid

Add a `GridViewPager` element to your layout definition as follows:

```
<android.support.wearable.view.GridViewPager
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/pager"
```

```
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

You can use any of the techniques described in Defining Layouts to ensure that your 2D picker works on both round and square devices.

Implement a Page Adapter

A page adapter provides a set of pages to populate a `GridViewPager` component. To implement this adapter, you extend the `FragmentGridPagerAdapter` class from the Wearable UI Library

The following snippet shows how to provide a set of static cards with custom background images:

```
public class SampleGridPagerAdapter extends FragmentGridPagerAdapter
{

    private final Context mContext;
    private List mRows;

    public SampleGridPagerAdapter(Context ctx, FragmentManager fm) {
        super(fm);
        mContext = ctx;
    }

    static final int[] BG_IMAGES = new int[] {
        R.drawable.debug_background_1, ...
        R.drawable.debug_background_5
    };

    // A simple container for static data in each page
    private static class Page {
        // static resources
        int titleRes;
        int textRes;
        int iconRes;
        ...
    }

    // Create a static set of pages in a 2D array
    private final Page[][] PAGES = { ... };

    // Override methods in FragmentGridPagerAdapter
    ...
}
```

The adapter calls `getFragment()` and `getBackgroundForRow()` to retrieve the content to display for each row:

```
// Obtain the UI fragment at the specified position
```

```
@Override
public Fragment getFragment(int row, int col) {
    Page page = PAGES[row][col];
    String title =
        page.titleRes != 0 ? mContext.getString(page.titleRes) :
null;
    String text =
        page.textRes != 0 ? mContext.getString(page.textRes) : null;
    CardFragment fragment = CardFragment.create(title, text,
page.iconRes);

    // Advanced settings (card gravity, card expansion/scrolling)
    fragment.setCardGravity(page.cardGravity);
    fragment.setExpansionEnabled(page.expansionEnabled);
    fragment.setExpansionDirection(page.expansionDirection);
    fragment.setExpansionFactor(page.expansionFactor);
    return fragment;
}

// Obtain the background image for the row
@Override
public Drawable getBackgroundForRow(int row) {
    return mContext.getResources().getDrawable(
            (BG_IMAGES[row % BG_IMAGES.length]), null);
}
```

The following example shows how to retrieve the background to display for a specific page in the grid:

```
// Obtain the background image for the specific page
@Override
public Drawable getBackgroundForPage(int row, int column) {
    if( row == 2 && column == 1) {
        // Place image at specified position
        return
mContext.getResources().getDrawable(R.drawable.bugdroid_large, null);
    } else {
        // Default to background image for row
        return GridPagerAdapter.BACKGROUND_NONE;
    }
}
```

The `getRowCount()` method tells the adapter how many rows of content are available, and the `getColumnCount()` method tells the adapter how many columns of content are available for each of the rows.

```
// Obtain the number of pages (vertical)
@Override
public int getRowCount() {
    return PAGES.length;
}

// Obtain the number of pages (horizontal)
@Override
public int getColumnCount(int rowNum) {
```

```
    return PAGES[rowNum].length;
}
```

The adapter implementation details depend on your particular set of pages. Each page provided by the adapter is of type `Fragment`. In this example, each page is a `CardFragment` instance that uses one of the default card layouts. However, you can combine different types of pages in the same 2D picker, such as cards, action icons, and custom layouts depending on your use cases.

Not all rows need to have the same number of pages. Notice that in this example the number of colums is different for each row. You can also use a `GridViewPager` component to implement a 1D picker with only one row or only one column.

The `GridViewPager` class provides support for scrolling in cards whose content does not fit the device screen. This example configures each card to expand as required, so users can scroll through the card's content. When users reach the end of a scrollable card, a swipe in the same direction shows the next page on the grid, if one is available.

Figure 1: The *GridViewPager* sample.

You can specify a custom background for each page with the `getBackgroundForPage()` method. When users swipe to navigate across pages, the `GridViewPager` class applies parallax and crossfade effects between different backgrounds automatically.

Assign an adapter instance to the page grid

In your activity, assign an instance of your adapter implementation to the `GridViewPager` component:

```
public class MainActivity extends Activity {

    @Override
```

```
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ...
        final GridViewPager pager = (GridViewPager)
findViewById(R.id.pager);
        pager.setAdapter(new SampleGridPagerAdapter(this,
getFragmentManager()));
    }
}
```

Next Get started Next class

Showing Confirmations

Previous Next Get started

This lesson teaches you to

1. Use Automatic Confirmation Timers
2. Show Confirmation Animations

You should also read

- Android Wear Design Principles

Confirmations in Android Wear apps use the whole screen or a larger portion of it than those in handheld apps. This ensures that users can see these confirmations by just glancing at the screen and that they have large enough touch targets to cancel an action.

The Wearable UI Library helps you show confirmation animations and timers in your Android Wear apps:

Confirmation timers
: Automatic confirmation timers show users an animated timer that lets them cancel an action they just performed.

Confirmation animations
: Confirmation animations give users visual feedback when they complete an action.

The following sections show you how to implement these patterns.

Use Automatic Confirmation Timers

Automatic confirmation timers let users

cancel an action they just performed. When the user performs the action, your app shows a button to cancel the action with a timer animation and starts the timer. The user has the option to cancel the action until the timer finishes. Your app gets notified if the user cancels the action and when the timer expires.

Figure 1: A confirmation timer.

To show a confirmation timer when users complete an action in your app:

1. Add a `<DelayedConfirmationView>` element to your layout.
2. Implement the `DelayedConfirmationListener` interface in your activity.
3. Set the duration of the timer and start it when the user completes an action.

Add the `<DelayedConfirmationView>` element to your layout as follows:

```
<android.support.wearable.view.DelayedConfirmationView
    android:id="@+id/delayed_confirm"
    android:layout_width="40dp"
    android:layout_height="40dp"
    android:src="@drawable/cancel_circle"
    app:circle_border_color="@color/lightblue"
    app:circle_border_width="4dp"
    app:circle_radius="16dp">
</android.support.wearable.view.DelayedConfirmationView>
```

You can assign a drawable resource to display inside the circle with the `android:src` attribute and configure the parameters of the circle directly on the layout definition.

To be notified when the timer finishes or when users tap on it, implement the corresponding listener methods in your activity:

```
public class WearActivity extends Activity implements

DelayedConfirmationView.DelayedConfirmationListener {

    private DelayedConfirmationView mDelayedView;
```

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_wear_activity);

        mDelayedView =
                (DelayedConfirmationView)
findViewById(R.id.delayed_confirm);
        mDelayedView.setListener(this);
    }

    @Override
    public void onTimerFinished(View view) {
        // User didn't cancel, perform the action
    }

    @Override
    public void onTimerSelected(View view) {
        // User canceled, abort the action
    }
}
```

To start the timer, add the following code to the point in your activity where users select an action:

```
// Two seconds to cancel the action
mDelayedView.setTotalTimeMs(2000);
// Start the timer
mDelayedView.start();
```

Show Confirmation Animations

To show a confirmation animation when users complete an action in your app, create an intent that starts `ConfirmationActivity` from one of your activities. You can specify one of the these animations with the `EXTRA_ANIMATION_TYPE` intent extra:

Figure 2: A confirmation animation.

- `SUCCESS_ANIMATION`
- `FAILURE_ANIMATION`
- `OPEN_ON_PHONE_ANIMATION`

You can also add a message that appears under the confirmation icon.

To use the `ConfirmationActivity` in your app, first declare this activity in

your manifest file:

```
<manifest>
  <application>
    ...
    <activity

android:name="android.support.wearable.activity.ConfirmationActivity">

    </activity>
  </application>
</manifest>
```

Then determine the result of the user action and start the activity with an intent:

```
Intent intent = new Intent(this, ConfirmationActivity.class);
intent.putExtra(ConfirmationActivity.EXTRA_ANIMATION_TYPE,
                ConfirmationActivity.SUCCESS_ANIMATION);
intent.putExtra(ConfirmationActivity.EXTRA_MESSAGE,
                getString(R.string.msg_sent));
startActivity(intent);
```

After showing the confirmation animation, `ConfirmationActivity` finishes and your activity resumes.

Next Get started Next class

Exiting Full-Screen Activities

Previous Next Get started

This lesson teaches you to

1. Disable the Swipe-To-Dismiss Gesture
2. Implement the Long Press to Dismiss Pattern

You should also read

- Android Wear Design Principles

By default, users exit Android Wear activities by swiping from left to right. If the app contains horizontally scrollable content, users first have to navigate to the edge of the content and then swipe again from left to right to exit the app.

For more immersive experiences, like an app that can scroll a map in any direction, you can disable the swipe to exit gesture in your app. However, if you disable it, you must implement the long-press-to-dismiss UI pattern to let users exit your app using the `DismissOverlayView` class from the Wearable UI Library. You must also inform your users the first time they run your app that they can exit using a long press.

For design guidelines about exiting Android Wear activities, see Breaking out of the card.

Disable the Swipe-To-Dismiss Gesture

If the user interaction model of your app interferes with the swipe-to-dismiss gesture, you can disable it for your app. To disable the swipe-to-dismiss gesture in your app, extend the default theme and set the

`android:windowSwipeToDismiss` attribute to `false`:

```
<style name="AppTheme" parent="Theme.DeviceDefault">
    <item name="android:windowSwipeToDismiss">false</item>
</style>
```

If you disable this gesture, you must implement the long-press-to-dismiss UI pattern to let users exit your app, as described in the next section.

Implement the Long Press to Dismiss Pattern

To use the `DismissOverlayView` class in your activity, add this element to your layout definition such that it covers the whole screen and is placed above all other views.

The following example shows how to add the `<DismissOverlayView>` element:

```
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_height="match_parent"
    android:layout_width="match_parent">

    <!-- other views go here -->

    <android.support.wearable.view.DismissOverlayView
        android:id="@+id/dismiss_overlay"
        android:layout_height="match_parent"
        android:layout_width="match_parent"/>
<FrameLayout>
```

In your activity, obtain the `<DismissOverlayView>` element and set some introductory text. This text is shown to users the first time they run your app to inform them that they can exit the app using a long press gesture. Then use a `GestureDetector` to detect a long press:

```
public class WearActivity extends Activity {

    private DismissOverlayView mDismissOverlay;
    private GestureDetector mDetector;

    public void onCreate(Bundle savedState) {
        super.onCreate(savedState);
        setContentView(R.layout.wear_activity);

        // Obtain the DismissOverlayView element
```

```
        mDismissOverlay = (DismissOverlayView)
findViewById(R.id.dismiss_overlay);
        mDismissOverlay.setIntroText(R.string.long_press_intro);
        mDismissOverlay.showIntroIfNecessary();

        // Configure a gesture detector
        mDetector = new GestureDetector(this, new
SimpleOnGestureListener() {
            public void onLongPress(MotionEvent ev) {
                mDismissOverlay.show();
            }
        });
    }

    // Capture long presses
    @Override
    public boolean onTouchEvent(MotionEvent ev) {
        return mDetector.onTouchEvent(ev) || super.onTouchEvent(ev);
    }
}
```

When the system detects a long press gesture, the `<DismissOverlayView>` element shows an **Exit** button, which terminates your activity if the user presses it.

Next Get started Next class

Sending and Syncing Data

Previous Next Get started

Dependencies and prerequisites

- Android 4.3 (API Level 18) or higher on the handset device
- The latest version of Google Play services
- An Android Wear device or Wear AVD

The Wearable Data Layer API, which is part of Google Play services, provides a communication channel for your handheld and wearable apps. The API consists of a set of data objects that the system can send and synchronize over the wire and listeners that notify your apps of important events with the data layer:

Data Items
: A `DataItem` provides data storage with automatic syncing between the handheld and wearable.

Messages
: The `MessageApi` class can send messages and is good for remote procedure calls (RPC), such as controlling a handheld's media player from the wearable or starting an intent on the wearable from the handheld. Messages are also great for one-way requests or for a request/response communication model. If the handheld and wearable are connected, the system queues the message for delivery and returns a successful result code. If the devices are not connected, an error is returned. A successful result code does not indicate that the message was delivered successfully as the devices may disconnect after receiving the result code.

Asset
: `Asset` objects are for sending binary blobs of data, such as images. You attach assets to data items and the system automatically takes

care of the transfer for you, conserving Bluetooth bandwidth by caching large assets to avoid re-transmission.

WearableListenerService (for services)

Extending `WearableListenerService` lets you listen for important data layer events in a service. The system manages the lifecycle of the `WearableListenerService`, binding to the service when it needs to send data items or messages and unbinding the service when no work is needed.

DataListener (for foreground activities)

Implementing `DataListener` in an activity lets you listen for important data layer events when an activity is in the foreground. Using this instead of the `WearableListenerService` lets you listen for changes only when the user is actively using your app.

Channel

You can use the `ChannelApi` class to transfer large data items, such as music and movie files, from a handheld to a wearable device. The Channel API for data transfer has the following benefits:

- Transfer large data files between two or more connected devices, without the automatic synchronization provided when using `Asset` objects attached to `DataItem` objects. The Channel API saves disk space unlike the `DataApi` class, which creates a copy of the assets on the local device before synchronizing with connected devices.
- Reliably send a file that is too large in size to send using the `MessageApi` class.
- Transfer streamed data, such as music pulled from a network server or voice data from the microphone.

Warning: Because these APIs are designed for communication between handhelds and wearables, these are the only APIs you should use to set up communication between these devices. For instance, don't try to open low-level sockets to create a communication channel.

Android Wear supports multiple wearables connected to a handheld device. For example, when the user saves a note on a handheld, it

automatically appears on both of the user's Wear devices. To synchronize data between devices, Google's servers host a cloud node in the network of devices. The system synchronizes data to directly connected devices, the cloud node, and to wearable devices connected to the cloud node via Wi-Fi.

Figure 1. A sample network of nodes with handheld and wearable devices.

Lessons

Accessing the Wearable Data Layer
: This lesson shows you how to create a client to access the Data Layer APIs.

Syncing Data Items
: Data items are objects that are stored in a replicated data store that is automatically synced from handhelds to wearables.

Transferring Assets
: Assets are binary blobs of data that you typically use to transfer

images or media.

Sending and Receiving Messages

Messages are designed for fire-and-forget messages that you can send back and forth between the wearable and handheld.

Handling Data Layer Events

Be notified of changes and events to the data layer.

Next Get started Next class

Accessing the Wearable Data Layer

Previous Next Get started

This lesson teaches you to

1. Set up a Google Play services client to use the Wearable Data Layer APIs

Dependencies and Prerequisites

1. Creating Wearable Apps > Set Up an Android Wear Emulator or Device
2. Creating Wearable Apps > Creating a Project

To call the Data Layer API, create an instance of `GoogleApiClient`, the main entry point for any of the Google Play services APIs.

`GoogleApiClient` provides a builder that makes it easy to create an instance of the client. A minimal `GoogleApiClient` looks like this:

Note: For now, this minimal client is enough to get started. However, see Accessing Google Play services APIs for more information about creating a `GoogleApiClient`, implementing its callbacks, and handling error cases.

```
GoogleApiClient mGoogleApiClient = new GoogleApiClient.Builder(this)
        .addConnectionCallbacks(new ConnectionCallbacks() {
                @Override
                public void onConnected(Bundle connectionHint) {
                    Log.d(TAG, "onConnected: " + connectionHint);
                    // Now you can use the Data Layer API
                }
                @Override
                public void onConnectionSuspended(int cause) {
                    Log.d(TAG, "onConnectionSuspended: " + cause);
```

```
                }
        })
        .addOnConnectionFailedListener(new
OnConnectionFailedListener() {
                @Override
                public void onConnectionFailed(ConnectionResult
result) {
                    Log.d(TAG, "onConnectionFailed: " + result);
                }
            })
        // Request access only to the Wearable API
        .addApi(Wearable.API)
        .build();
```

Important: If you are adding multiple APIs to a `GoogleApiClient`, you may run into client connection errors on devices that do not have the Android Wear app installed. To avoid connection errors, call the `addApiIfAvailable()` method and pass in the `Wearable` API to indicate that your client should gracefully handle the missing API. For more information, see Access the Wearable API.

Before you use the data layer API, start a connection on your client by calling the `connect()` method, as described in Start a Connection. When the system invokes the `onConnected()` callback for your client, you're ready to use the Data Layer API.

Next Get started Next class

Syncing Data Items

Previous Next Get started

This lesson teaches you to

1. Sync Data with a Data Map
2. Listen for Data Item Events

A `DataItem` defines the data interface that the system uses to synchronize data between handhelds and wearables. A `DataItem` generally consists of the following items:

- **Payload** - A byte array, which you can set with whatever data you wish, allowing you to do your own object serialization and deserialization. The size of the payload is limited to 100KB.
- **Path** - A unique string that must start with a forward slash (for instance, `"/path/to/data"`)

You normally don't implement `DataItem` directly. Instead, you:

1. Create a `PutDataRequest` object, specifying a string path to uniquely identify the item.
2. Call `setData()` to set the payload.
3. Call `DataApi.putDataItem()` to request the system to create the data item.
4. When requesting data items, the system returns objects that properly implement the `DataItem` interface.

However, instead of working with raw bytes using `setData()`, we recommend you use a data map, which exposes a data item in an easy-to-use `Bundle`-like interface.

Sync Data with a Data Map

When possible, use the `DataMap` class. This approach lets you work with data items in the form of an Android `Bundle`, so object serialization and de-serialization is done for you, and you can manipulate data with key-value pairs.

To use a data map:

1. Create a `PutDataMapRequest` object, setting the path of the data item.

   Note: The path string is a unique identifier for the data item that allows you to access it from either side of the connection. The path must begin with a forward slash. If you're using hierarchical data in your app, you should create a path scheme that matches the structure of the data.

2. Call `PutDataMapRequest.getDataMap()` to obtain a data map that you can set values on.
3. Set any desired values for the data map using the `put...()` methods, such as `putString()`.
4. Call `PutDataMapRequest.asPutDataRequest()` to obtain a `PutDataRequest` object.
5. Call `DataApi.putDataItem()` to request the system to create the data item.

   Note: If the handset and wearable devices are disconnected, the data is buffered and synced when the connection is re-established.

The `increaseCounter()` method in the following example shows how to create a data map and put data in it:

```
public class MainActivity extends Activity implements
        DataApi.DataListener,
        GoogleApiClient.ConnectionCallbacks,
        GoogleApiClient.OnConnectionFailedListener {

    private static final String COUNT_KEY = "com.example.key.count";

    private GoogleApiClient mGoogleApiClient;
    private int count = 0;

    ...

    // Create a data map and put data in it
```

```
    private void increaseCounter() {
        PutDataMapRequest putDataMapReq =
PutDataMapRequest.create("/count");
        putDataMapReq.getDataMap().putInt(COUNT_KEY, count++);
        PutDataRequest putDataReq = putDataMapReq.asPutDataRequest();
        PendingResult<DataApi.DataItemResult> pendingResult =
                Wearable.DataApi.putDataItem(mGoogleApiClient,
putDataReq);
    }

    ...
}
```

For more information about handling the `PendingResult` object, see Wait for the Status of Data Layer Calls.

Listen for Data Item Events

If one side of the data layer connection changes a data item, you probably want to be notified of any changes on the other side of the connection. You can do this by implementing a listener for data item events.

The code snippet in the following example notifies your app when the value of the counter defined in the previous example changes:

```
public class MainActivity extends Activity implements
        DataApi.DataListener,
        GoogleApiClient.ConnectionCallbacks,
        GoogleApiClient.OnConnectionFailedListener {

    private static final String COUNT_KEY = "com.example.key.count";

    private GoogleApiClient mGoogleApiClient;
    private int count = 0;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        mGoogleApiClient = new GoogleApiClient.Builder(this)
                .addApi(Wearable.API)
                .addConnectionCallbacks(this)
                .addOnConnectionFailedListener(this)
                .build();
    }

    @Override
    protected void onResume() {
        super.onStart();
        mGoogleApiClient.connect();
```

```
    }

    @Override
    public void onConnected(Bundle bundle) {
        Wearable.DataApi.addListener(mGoogleApiClient, this);
    }

    @Override
    protected void onPause() {
        super.onPause();
        Wearable.DataApi.removeListener(mGoogleApiClient, this);
        mGoogleApiClient.disconnect();
    }

    @Override
    public void onDataChanged(DataEventBuffer dataEvents) {
        for (DataEvent event : dataEvents) {
            if (event.getType() == DataEvent.TYPE_CHANGED) {
                // DataItem changed
                DataItem item = event.getDataItem();
                if (item.getUri().getPath().compareTo("/count") == 0)
{
                    DataMap dataMap =
DataMapItem.fromDataItem(item).getDataMap();
                    updateCount(dataMap.getInt(COUNT_KEY));
                }
            } else if (event.getType() == DataEvent.TYPE_DELETED) {
                // DataItem deleted
            }
        }
    }

    // Our method to update the count
    private void updateCount(int c) { ... }

    ...
}
```

This activity implements the `DataItem.DataListener` interface. This activity adds itself as a listener for data item events inside the `onConnected()` method and removes the listener in the `onPause()` method.

You can also implement the listener as a service. For more information, see Listen for Data Layer Events.

Next Get started Next class

Transferring Assets

Previous Next Get started

This lesson teaches you to

1. Transfer an Asset
2. Receive an Asset

To send large blobs of binary data over the Bluetooth transport, such as images, attach an `Asset` to a data item and the put the data item into the replicated data store.

Assets automatically handle caching of data to prevent retransmission and conserve Bluetooth bandwidth. A common pattern is for a handheld app to download an image, shrink it to an appropriate size for display on the wearable, and transmit it to the wearable app as an asset. The following examples demonstrate this pattern.

Note: Although the size of data items is limited to 100KB, assets can be as large as desired. However, transferring large assets affects the user experience in many cases, so test your apps to ensure that they perform well if you're transferring large assets.

Transfer an Asset

Create the asset using one of the `create...()` methods in the `Asset` class. Here, we convert a bitmap to a byte stream and then call `createFromBytes()` to create the asset.

```
private static Asset createAssetFromBitmap(Bitmap bitmap) {
    final ByteArrayOutputStream byteStream = new
ByteArrayOutputStream();
    bitmap.compress(Bitmap.CompressFormat.PNG, 100, byteStream);
    return Asset.createFromBytes(byteStream.toByteArray());
}
```

When you have an asset, attach it to a data item with the `putAsset()` method in `DataMap` or `PutDataRequest` and then put the data item into the data store with `putDataItem()`:

Using PutDataRequest

```
Bitmap bitmap = BitmapFactory.decodeResource(getResources(),
R.drawable.image);
Asset asset = createAssetFromBitmap(bitmap);
PutDataRequest request = PutDataRequest.create("/image");
request.putAsset("profileImage", asset);
Wearable.DataApi.putDataItem(mGoogleApiClient, request);
```

Using PutDataMapRequest

```
Bitmap bitmap = BitmapFactory.decodeResource(getResources(),
R.drawable.image);
Asset asset = createAssetFromBitmap(bitmap);
PutDataMapRequest dataMap = PutDataMapRequest.create("/image");
dataMap.getDataMap().putAsset("profileImage", asset)
PutDataRequest request = dataMap.asPutDataRequest();
PendingResult<DataApi.DataItemResult> pendingResult =
Wearable.DataApi
        .putDataItem(mGoogleApiClient, request);
```

Receive assets

When an asset is created, you probably want to read and extract it on other side of the connection. Here's an example of how to implement the callback to detect an asset change and extract the asset:

```
@Override
public void onDataChanged(DataEventBuffer dataEvents) {
  for (DataEvent event : dataEvents) {
    if (event.getType() == DataEvent.TYPE_CHANGED &&
        event.getDataItem().getUri().getPath().equals("/image")) {
      DataMapItem dataMapItem =
DataMapItem.fromDataItem(event.getDataItem());
      Asset profileAsset =
dataMapItem.getDataMap().getAsset("profileImage");
      Bitmap bitmap = loadBitmapFromAsset(profileAsset);
      // Do something with the bitmap
    }
  }
}

public Bitmap loadBitmapFromAsset(Asset asset) {
    if (asset == null) {
        throw new IllegalArgumentException("Asset must be non-
null");
    }
```

```
    ConnectionResult result =
            mGoogleApiClient.blockingConnect(TIMEOUT_MS,
TimeUnit.MILLISECONDS);
    if (!result.isSuccess()) {
        return null;
    }
    // convert asset into a file descriptor and block until it's
ready
    InputStream assetInputStream = Wearable.DataApi.getFdForAsset(
            mGoogleApiClient, asset).await().getInputStream();
            mGoogleApiClient.disconnect();

    if (assetInputStream == null) {
        Log.w(TAG, "Requested an unknown Asset.");
        return null;
    }
    // decode the stream into a bitmap
    return BitmapFactory.decodeStream(assetInputStream);
}
```

Next Get started Next class

Sending and Receiving Messages

Previous Next Get started

This lesson teaches you to

1. Send a Message
2. Receive a Message

Try it out

- FindMyPhone

You send messages using the `MessageApi` and attach the following items to the message:

- An arbitrary payload (optional)
- A path that uniquely identifies the message's action

Unlike with data items, there is no syncing between the handheld and wearable apps. Messages are a one-way communication mechanism that's good for remote procedure calls (RPC), such as sending a message to the wearable to start an activity.

Multiple wearable devices can be connected to a user's handheld device. Each connected device in the network is considered a *node*. With multiple connected devices, you must consider which nodes receive the messages. For example, in a voice transcription app that receives voice data on the wearable device, you should send the message to a node with the processing power and battery capacity to handle the request, such as a handheld device.

Note: With versions of Google Play services prior to 7.3.0, only one wearable device could be connected to a handheld device at a time. You

may need to update your existing code to take the multiple connected nodes feature into consideration. If you don't implement the changes, your messages may not get delivered to intended devices.

Send a Message

A wearable app can provide functionality for users such as voice transcription. Users can speak into their wearable device's microphone, and have a transcription saved to a note. Since a wearable device typically does not have the processing power and battery capacity required to handle the voice transcription activity, the app should offload this work to a more capable, connected device.

The following sections show you how to advertise device nodes that can process activity requests, discover the nodes capable of fulfilling a requested need, and send messages to those nodes.

Advertise capabilities

To launch an activity on a handheld device from a wearable device, use the `MessageApi` class to send the request. Since multiple wearables can be connected to the handheld device, the wearable app needs to determine that a connected node is capable of launching the activity. In your handheld app, advertise that the node it runs on provides specific capabilities.

To advertise the capabilities of your handheld app:

1. Create an XML configuration file in the `res/values/` directory of your project and name it `wear.xml`.
2. Add a resource named `android_wear_capabilities` to `wear.xml`.
3. Define capabilities that the device provides.

Note: Capabilities are custom strings that you define and must be unique within your app.

The following example shows how to add a capability named `voice_transcription` to `wear.xml`:

```
<resources>
    <string-array name="android_wear_capabilities">
        <item>voice_transcription</item>
    </string-array>
</resources>
```

Retrieve the nodes with the required capabilities

Initially, you can detect the capable nodes by calling the `CapabilityApi.getCapability()` method. The following example shows how to manually retrieve the results of reachable nodes with the `voice_transcription` capability:

```
private static final String
        VOICE_TRANSCRIPTION_CAPABILITY_NAME = "voice_transcription";

private GoogleApiClient mGoogleApiClient;

...

private void setupVoiceTranscription() {
    CapabilityApi.GetCapabilityResult result =
            Wearable.CapabilityApi.getCapability(
                    mGoogleApiClient,
VOICE_TRANSCRIPTION_CAPABILITY_NAME,
                    CapabilityApi.FILTER_REACHABLE).await();

    updateTranscriptionCapability(result.getCapability());
}
```

To detect capable nodes as they connect to the wearable device, register a `CapabilityApi.CapabilityListener()` instance to your `GoogleApiClient`. The following example shows how to register the listener and retrieve the results of reachable nodes with the `voice_transcription` capability:

```
private void setupVoiceTranscription() {
    ...

    CapabilityApi.CapabilityListener capabilityListener =
            new CapabilityApi.CapabilityListener() {
                @Override
                public void onCapabilityChanged(CapabilityInfo
capabilityInfo) {
                    updateTranscriptionCapability(capabilityInfo);
                }
            };

    Wearable.CapabilityApi.addCapabilityListener(
            mGoogleApiClient,
            capabilityListener,
            VOICE_TRANSCRIPTION_CAPABILITY_NAME);
}
```

Note: If you create a service that extends `WearableListenerService` to detect capability changes, you may want to override the `onConnectedNodes()` method to listen to finer-grained connectivity details, such as when a wearable device switches from Wi-Fi to a Bluetooth connection to the handset. For an example implementation, see the `DisconnectListenerService` class in the FindMyPhone sample. For more information on how to listen for important events, see Listen for Data Layer Events.

After detecting the capable nodes, determine where to send the message. You should pick a node that is in close proximity to your wearable device to minimize message routing through multiple nodes. A nearby node is defined as one that is directly connected to the device. To determine if a node is nearby, call the `Node.isNearby()` method.

The following example shows how you might determine the best node to use:

```
private String transcriptionNodeId = null;

private void updateTranscriptionCapability(CapabilityInfo
capabilityInfo) {
    Set<Node> connectedNodes = capabilityInfo.getNodes();

    transcriptionNodeId = pickBestNodeId(connectedNodes);
}

private String pickBestNodeId(Set<Node> nodes) {
    String bestNodeId = null;
    // Find a nearby node or pick one arbitrarily
    for (Node node : nodes) {
        if (node.isNearby()) {
            return node.getId();
         }
         bestNodeId = node.getId();
    }
    return bestNodeId;
}
```

Deliver the message

Once you've identified the best node to use, send the message using the `MessageApi` class.

The following example shows how to send a message to the

transcription-capable node from a wearable device. Verify that the node is available before you attempt to send the message. This call is synchronous and blocks processing until the system queues the message for delivery.

Note: A successful result code does not guarantee delivery of the message. If your app requires data reliability, use `DataItem` objects or the `ChannelApi` class to send data between devices.

```
public static final String VOICE_TRANSCRIPTION_MESSAGE_PATH =
"/voice_transcription";

private void requestTranscription(byte[] voiceData) {
    if (transcriptionNodeId != null) {
        Wearable.MessageApi.sendMessage(googleApiClient,
transcriptionNodeId,
            VOICE_TRANSCRIPTION_MESSAGE_PATH,
voiceData).setResultCallback(
                  new ResultCallback() {
                      @Override
                      public void onResult(SendMessageResult
sendMessageResult) {
                          if
(!sendMessageResult.getStatus().isSuccess()) {
                              // Failed to send message
                          }
                      }
                  }
            );
    } else {
        // Unable to retrieve node with transcription capability
    }
}
```

Note: To learn more about asynchronous and synchronous calls to Google Play services and when to use each, see Communicate with Google Play Services.

You can also broadcast messages to all connected nodes. To retrieve all of the connected nodes that you can send messages to, implement the following code:

```
private Collection<String> getNodes() {
    HashSet <String>results = new HashSet<String>();
    NodeApi.GetConnectedNodesResult nodes =

Wearable.NodeApi.getConnectedNodes(mGoogleApiClient).await();
    for (Node node : nodes.getNodes()) {
        results.add(node.getId());
    }
```

```
    return results;
}
```

Receive a Message

To be notified of received messages, implement the `MessageListener` interface to provide a listener for message events. Then, register the listener with the `MessageApi.addListener()` method. This example shows how you might implement the listener to check the `VOICE_TRANSCRIPTION_MESSAGE_PATH`. If this condition is `true`, start an activity to process the voice data.

```
@Override
public void onMessageReceived(MessageEvent messageEvent) {
    if
(messageEvent.getPath().equals(VOICE_TRANSCRIPTION_MESSAGE_PATH)) {
        Intent startIntent = new Intent(this, MainActivity.class);
        startIntent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
        startIntent.putExtra("VOICE_DATA", messageEvent.getData());
        startActivity(startIntent);
    }
}
```

This is just a snippet that requires more implementation details. Learn about how to implement a full listener service or activity in Listening for Data Layer Events.

Next Get started Next class

Handling Data Layer Events

Previous Next Get started

This lesson teaches you to

1. Wait for the Status of Data Layer Calls
2. Listen for Data Layer Events

When you make calls to the Data Layer API, you can receive the status of the call when it completes as well as listen for any changes that the call ends up making with listeners.

Wait for the Status of Data Layer Calls

You'll notice that calls to the Data Layer API sometimes return a `PendingResult`, such as `putDataItem()`. As soon as the `PendingResult` is created, the operation is queued in the background. If you do nothing else after this, the operation eventually completes silently. However, you'll usually want to do something with the result after the operation completes, so the `PendingResult` lets you wait for the result status, either synchronously or asynchronously.

Asynchronous calls

If your code is running on the main UI thread, do not make blocking calls to the Data Layer API. You can run the calls asynchronously by adding a callback method to the `PendingResult` object, which fires when the operation is completed:

```
pendingResult.setResultCallback(new ResultCallback<DataItemResult>()
{
    @Override
    public void onResult(final DataItemResult result) {
        if(result.getStatus().isSuccess()) {
            Log.d(TAG, "Data item set: " +
```

```
result.getDataItem().getUri());
        }
    }
});
```

Synchronous calls

If your code is running on a separate handler thread in a background service (which is the case in a `WearableListenerService`), it's fine for the calls to block. In this case, you can call `await()` on the `PendingResult` object, which blocks until the request completes and returns a `Result` object:

```
DataItemResult result = pendingResult.await();
if(result.getStatus().isSuccess()) {
    Log.d(TAG, "Data item set: " + result.getDataItem().getUri());
}
```

Listen for Data Layer Events

Because the data layer synchronizes and sends data across the handheld and wearable, you normally want to listen for important events, such as when data items are created, messages are received, or when the wearable and handset are connected.

To listen for data layer events, you have two options:

- Create a service that extends `WearableListenerService`.
- Create an activity that implements `DataApi.DataListener`.

With both these options, you override the data event callback methods for the events you are interested in handling.

With a WearableListenerService

You typically create instances of this service in both your wearable and handheld apps. If you are not interested in data events in one of these apps, then you don't need to implement this service in that particular app.

For example, you can have a handheld app that sets and gets data item

objects and a wearable app that listens for these updates to update it's UI. The wearable never updates any of the data items, so the handheld app doesn't listen for any data events from the wearable app.

You can listen for the following events with `WearableListenerService`:

- `onDataChanged()` - Called when data item objects are created, changed, or deleted. An event on one side of a connection triggers this callback on both sides.
- `onMessageReceived()` - A message sent from one side of a connection triggers this callback on the other side of the connection.
- `onPeerConnected()` and `onPeerDisconnected()` - Called when the connection with the handheld or wearable is connected or disconnected. Changes in connection state on one side of the connection trigger these callbacks on both sides of the connection.

To create a `WearableListenerService`:

1. Create a class that extends `WearableListenerService`.
2. Listen for the events that you're interested in, such as `onDataChanged()`.
3. Declare an intent filter in your Android manifest to notify the system about your `WearableListenerService`. This allows the system to bind your service as needed.

The following example shows how to implement a simple `WearableListenerService`:

```
public class DataLayerListenerService extends WearableListenerService
{

    private static final String TAG = "DataLayerSample";
    private static final String START_ACTIVITY_PATH = "/start-
activity";
    private static final String DATA_ITEM_RECEIVED_PATH = "/data-
item-received";

    @Override
    public void onDataChanged(DataEventBuffer dataEvents) {
        if (Log.isLoggable(TAG, Log.DEBUG)) {
            Log.d(TAG, "onDataChanged: " + dataEvents);
        }
        final List events = FreezableUtils
                .freezeIterable(dataEvents);
```

```
        GoogleApiClient googleApiClient = new
GoogleApiClient.Builder(this)
                .addApi(Wearable.API)
                .build();

        ConnectionResult connectionResult =
                googleApiClient.blockingConnect(30,
TimeUnit.SECONDS);

        if (!connectionResult.isSuccess()) {
            Log.e(TAG, "Failed to connect to GoogleApiClient.");
            return;
        }

        // Loop through the events and send a message
        // to the node that created the data item.
        for (DataEvent event : events) {
            Uri uri = event.getDataItem().getUri();

            // Get the node id from the host value of the URI
            String nodeId = uri.getHost();
            // Set the data of the message to be the bytes of the
URI
            byte[] payload = uri.toString().getBytes();

            // Send the RPC
            Wearable.MessageApi.sendMessage(googleApiClient, nodeId,
                    DATA_ITEM_RECEIVED_PATH, payload);
        }
    }
}
```

Here's the corresponding intent filter in the Android manifest file:

```
<service android:name=".DataLayerListenerService">
  <intent-filter>
      <action
android:name="com.google.android.gms.wearable.BIND_LISTENER" />
  </intent-filter>
</service>
```

Permissions within Data Layer Callbacks

To deliver callbacks to your application for data layer events, Google Play services binds to your `WearableListenerService`, and calls your callbacks via IPC. This has the consequence that your callbacks inherit the permissions of the calling process.

If you try to perform a privileged operation within a callback, the security check fails because your callback is running with the identity of the calling process, instead of the identity of your app's process.

To fix this, call `clearCallingIdentity()` , to reset identity after crossing the IPC boundary, and then restore identity with `restoreCallingIdentity()` when you've completed the privileged operation:

```
long token = Binder.clearCallingIdentity();
try {
    performOperationRequiringPermissions();
} finally {
    Binder.restoreCallingIdentity(token);
}
```

With a Listener Activity

If your app only cares about data layer events when the user is interacting with the app and does not need a long-running service to handle every data change, you can listen for events in an activity by implementing one or more of the following interfaces:

- `DataApi.DataListener`
- `MessageApi.MessageListener`
- `NodeApi.NodeListener`

To create an activity that listens for data events:

1. Implement the desired interfaces.
2. In `onCreate(Bundle)`, create an instance of `GoogleApiClient` to work with the Data Layer API.
3. In `onStart()`, call `connect()` to connect the client to Google Play services.
4. When the connection to Google Play services is established, the system calls `onConnected()`. This is where you call `DataApi.addListener()`, `MessageApi.addListener()`, or `NodeApi.addListener()` to notify Google Play services that your activity is interested in listening for data layer events.
5. In `onStop()`, unregister any listeners with `DataApi.removeListener()`, `MessageApi.removeListener()`, or `NodeApi.removeListener()`.
6. Implement `onDataChanged(), onMessageReceived()`, `onPeerConnected()`, and `onPeerDisconnected()`, depending on the interfaces that you implemented.

Here's an example that implements DataApi.DataListener:

```
public class MainActivity extends Activity implements
        DataApi.DataListener, ConnectionCallbacks,
OnConnectionFailedListener {

    private GoogleApiClient mGoogleApiClient;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.main);
        mGoogleApiClient = new GoogleApiClient.Builder(this)
                .addApi(Wearable.API)
                .addConnectionCallbacks(this)
                .addOnConnectionFailedListener(this)
                .build();
    }

    @Override
    protected void onStart() {
        super.onStart();
        if (!mResolvingError) {
            mGoogleApiClient.connect();
        }
    }

    @Override
    public void onConnected(Bundle connectionHint) {
        if (Log.isLoggable(TAG, Log.DEBUG)) {
            Log.d(TAG, "Connected to Google Api Service");
        }
        Wearable.DataApi.addListener(mGoogleApiClient, this);
    }

    @Override
    protected void onStop() {
        if (null != mGoogleApiClient &&
mGoogleApiClient.isConnected()) {
            Wearable.DataApi.removeListener(mGoogleApiClient, this);
            mGoogleApiClient.disconnect();
        }
        super.onStop();
    }

    @Override
    public void onDataChanged(DataEventBuffer dataEvents) {
        for (DataEvent event : dataEvents) {
            if (event.getType() == DataEvent.TYPE_DELETED) {
                Log.d(TAG, "DataItem deleted: " +
event.getDataItem().getUri());
            } else if (event.getType() == DataEvent.TYPE_CHANGED) {
                Log.d(TAG, "DataItem changed: " +
event.getDataItem().getUri());
            }
        }
    }
}
```

Next Get started Next class

Creating Watch Faces

Previous Next Get started

Dependencies and Prerequisites

- Android 4.3 (API level 18) or higher on the handheld device
- Android 5.0 (API level 21) or higher on the wearable device

Design Guide

Watch Faces

Video

DevBytes: Watch Faces for Android Wear

Watch faces in Android Wear leverage a dynamic digital canvas to tell time using colors, animations, and relevant contextual information. The Android Wear companion app provides watch faces with different styles and shapes. When users select one of the available watch faces on the wearable or on the companion app, the wearable device previews the watch face and lets the user set configuration options.

Android Wear enables you to create custom watch faces for Wear devices. When users install a handheld app that contains a wearable app with watch faces, they become available in the Android Wear companion app on the handheld device and in the watch face picker on the wearable device.

This class teaches you to implement custom watch faces and to package them inside a wearable app. This class also covers design considerations and implementation tips to ensure that your designs integrate with system UI elements and are power-efficient.

Note: We recommend using Android Studio for Android Wear development as it provides project setup, library inclusion, and packaging conveniences that aren't available in the Eclipse Android Developer Tools. This training assumes you are using Android Studio.

Lessons

Designing Watch Faces
: Learn how to design a watch face that works on any Android Wear device.

Building a Watch Face Service
: Learn how to respond to important events during the lifecycle of your watch face.

Drawing Watch Faces
: Learn how to draw your watch face on a Wear device screen.

Showing Information in Watch Faces
: Learn how to incorporate contextual information into your watch face.

Providing Configuration Activities
: Learn how to create watch faces with configurable parameters.

Addressing Common Issues
: Learn how to fix common problems when developing a watch face.

Optimizing Performance and Battery Life
: Learn how to improve the frame rate of your animations and how to save power.

Next Get started Next class

Designing Watch Faces

Previous Next Get started

This lesson teaches you to

1. Conform to the Design Guidelines
2. Create an Implementation Strategy

You should also read

- Watch Faces for Android Wear

Design Guide

Watch Faces

Similar to the process of designing a traditional watch face, creating one for Android Wear is an exercise in visualizing time clearly. Android Wear devices provide advanced capabilities for watch faces that you can leverage in your designs, such as vibrant colors, dynamic backgrounds, animations, and data integration. However, there are also many design considerations that you must take into account.

This lesson provides a summary of the design considerations for watch faces and general guidelines to get started implementing a design. For more information, read the Watch Faces for Android Wear design guide.

Conform to the Design Guidelines

As you plan the look of your watch face and what kind of information it should present to users, consider these design guidelines:

Plan for square and round devices
Your design should work for both square and round Android Wear devices, including devices with insets on the bottom of the screen.

Support all display modes
Your watch face should support ambient mode with limited color and interactive mode with full color and animations.

Optimize for special screen technologies
In ambient mode, your watch face should keep most pixels black. Depending on the screen technology, you may need to avoid large blocks of white pixels, use only black and white, and disable anti-aliasing.

Figure 1. Example watch faces.

Accomodate system UI elements
Your design should ensure that system indicators remain visible and that users can still read the time when notification cards appear on the screen.

Integrate data
Your watch face can leverage sensors and cellular connectivity on the companion mobile device to show user data relevant to the context, such as the weather for the day or their next calendar event.

Provide configuration options
You can let users configure some aspects of your design (like colors and sizes) on the wearable or on the Android Wear companion app.

For more information about designing watch faces for Android Wear, see the Watch Faces for Android Wear design guide.

Create an Implementation Strategy

After you finalize the design for your watch face, you need to determine how to obtain any necessary data and draw the watch face on the wearable device. Most implementations consist of the following components:

- One or more background images
- Application code to retrieve the required data
- Application code to draw text and shapes over the background images

You typically use one background image in interactive mode and a different background image in ambient mode. The background in ambient mode is often completely black. Background images for Android Wear devices with a screen density of hdpi should be 320 by 320 pixels in size to fit both square and round devices. The corners of the background image are not visible on round devices. In your code, you can detect the size of the device screen and scale down the background image if the device has a lower resolution than your image. To improve performance, you should scale the background image only once and store the resulting bitmap.

You should run the application code to retrieve contextual data only as often as required and store the results to reuse the data every time you draw the watch face. For example, you don't need to fetch weather updates every minute.

To increase battery life, the application code that draws your watch face in ambient mode should be relatively simple. You usually draw outlines of shapes using a limited set of colors in this mode. In interactive mode, you can use full color, complex shapes, gradients, and animations to draw your watch face.

The remaining lessons in this class show you how to implement watch faces in detail.

Next Get started Next class

Building a Watch Face Service

Previous Next Get started

This lesson teaches you to

1. Create and Configure Your Project
2. Implement the Service Callback Methods
3. Register the Service Implementation

Related Samples

- WatchFace

You should also read

- Watch Faces for Android Wear

Watch faces in Android Wear are implemented as services and packaged inside a wearable app. When users install a handheld app that contains a wearable app with watch faces, these watch faces become available in the Android Wear companion app on the handheld device and in the watch face picker on the wearable. When users select one of the available watch faces, the wearable device shows the watch face and invokes its service callback methods as required.

This lesson shows you how to configure an Android project to include watch faces and how to implement the watch face service.

Create and Configure Your Project

To create an Android project for your watch face in Android Studio:

1. Start Android Studio.
2. Create a new project:
   - If you don't have a project opened, in the **Welcome** screen, click **New Project**.
   - If you have a project opened, from the **File** menu, select **New Project**.
3. Provide an application name and click **Next**.
4. Select the **Phone and Tablet** form factor.
5. Under **Minimum SDK**, choose API 18.
6. Select the **Wear** form factor.
7. Under **Minimum SDK**, choose API 21 and click **Next**.
8. Select **Add No Activity** and click **Next** in the two following screens.
9. Click **Finish**.
10. Click **View** > **Tool Windows** > **Project** in the IDE window.

Android Studio creates a project with the `wear` and `mobile` modules. For more information, see Creating a Project.

Dependencies

The Wearable Support Library provides the necessary classes that you extend to create watch face implementations. The Google Play services client libraries (`play-services` and `play-services-wearable`) are required to sync data items between the companion device and the wearable with the Wearable Data Layer API.

Android Studio automatically adds the required entries in your `build.gradle` files when you create the project in the instructions above.

Wearable Support Library API Reference

The reference documentation provides detailed information about the classes you use to implement watch faces. Browse the API reference documentation for the Wearable Support Library.

Download the Wearable Support Library for Eclipse ADT

If you are using the ADT plugin for Eclipse, download the Wearable Support Library and include it as a dependency in your project.

Declare Permissions

Watch faces require the `PROVIDE_BACKGROUND` and `WAKE_LOCK` permissions. Add the following permissions to the manifest files of both the wearable app and the mobile app under the `manifest` element:

```
<manifest ...>
    <uses-permission

android:name="com.google.android.permission.PROVIDE_BACKGROUND" />
    <uses-permission
        android:name="android.permission.WAKE_LOCK" />
    ...
</manifest>
```

Caution: The handheld app must include all the permissions declared in the wearable app.

Implement the Service and Callback Methods

Watch faces in Android Wear are implemented as services. When a watch face is active, the system invokes the methods in its service when the time changes or when an important event occurs (like switching to ambient mode or receiving a new notification). The service implementation then draws the watch face on the screen using the updated time and any other relevant data.

To implement a watch face, you extend the `CanvasWatchFaceService` and `CanvasWatchFaceService.Engine` classes, and then you override the callback methods in the `CanvasWatchFaceService.Engine` class. These classes are included in the Wearable Support Library.

The following snippet outlines the key methods you need to implement:

```
public class AnalogWatchFaceService extends CanvasWatchFaceService {

    @Override
    public Engine onCreateEngine() {
```

```
        /* provide your watch face implementation */
        return new Engine();
    }

    /* implement service callback methods */
    private class Engine extends CanvasWatchFaceService.Engine {

        @Override
        public void onCreate(SurfaceHolder holder) {
            super.onCreate(holder);
            /* initialize your watch face */
        }

        @Override
        public void onPropertiesChanged(Bundle properties) {
            super.onPropertiesChanged(properties);
            /* get device features (burn-in, low-bit ambient) */
        }

        @Override
        public void onTimeTick() {
            super.onTimeTick();
            /* the time changed */
        }

        @Override
        public void onAmbientModeChanged(boolean inAmbientMode) {
            super.onAmbientModeChanged(inAmbientMode);
            /* the wearable switched between modes */
        }

        @Override
        public void onDraw(Canvas canvas, Rect bounds) {
            /* draw your watch face */
        }

        @Override
        public void onVisibilityChanged(boolean visible) {
            super.onVisibilityChanged(visible);
            /* the watch face became visible or invisible */
        }
    }
}
```

The `CanvasWatchFaceService` class provides an invalidate mechanism similar to the `View.invalidate()` method. You can call the `invalidate()` method throughout your implementation when you want the system to redraw the watch face. You can only use the `invalidate()` method in the main UI thread. To invalidate the canvas from another thread, call the `postInvalidate()` method.

For more information about implementing the methods in the `CanvasWatchFaceService.Engine` class, see Drawing Watch Faces.

Register the Watch Face Service

After you implement the watch face service, you register the implementation in the manifest file of the wearable app. When users install this app, the system uses the information about the service to make the watch face available in the Android Wear companion app and in the watch face picker on the wearable device.

The following snippet shows how to register a watch face implementation under the `application` element:

```
<service
    android:name=".AnalogWatchFaceService"
    android:label="@string/analog_name"
    android:allowEmbedded="true"
    android:taskAffinity=""
    android:permission="android.permission.BIND_WALLPAPER" >
    <meta-data
        android:name="android.service.wallpaper"
        android:resource="@xml/watch_face" />
    <meta-data
        android:name="com.google.android.wearable.watchface.preview"
        android:resource="@drawable/preview_analog" />
    <meta-data

android:name="com.google.android.wearable.watchface.preview_circular"
        android:resource="@drawable/preview_analog_circular" />
    <intent-filter>
        <action
android:name="android.service.wallpaper.WallpaperService" />
        <category
            android:name=

"com.google.android.wearable.watchface.category.WATCH_FACE" />
    </intent-filter>
</service>
```

The Android Wear companion app and the watch face picker on the wearable device use the preview image defined by the `com.google.android.wearable.watchface.preview` metadata entry when presenting users with all the watch faces installed on the device. To obtain this drawable, run the watch face on your Android Wear device or in an emulator instance and take a screenshot. On Android Wear devices with hdpi screens, the preview image is typically 320x320 pixels in size.

Watch faces that look substantially different on round devices can provide both round and square preview images. To specify a round preview

image, use the `com.google.android.wearable.watchface.preview_circular` metadata entry. If a watch face includes both preview images, the companion app and the watch face picker on the wearable show the appropriate one, depending on the shape of the watch. If a round preview image is not included, the square preview image is used for both square and round devices. For round devices, a square preview image is cropped using a circular shape.

The `android.service.wallpaper` metadata entry specifies the `watch_face.xml` resource file, which contains a `wallpaper` element:

```
<?xml version="1.0" encoding="UTF-8"?>
<wallpaper xmlns:android="http://schemas.android.com/apk/res/android"
/>
```

Your wearable app can contain more than one watch face. You must add a service entry to the manifest file of the wearable app for each of your watch face implementations.

Next Get started Next class

Drawing Watch Faces

Previous Next Get started

This lesson teaches you to

1. Initialize Your Watch Face
2. Configure the System UI
3. Obtain Information About the Device Screen
4. Respond to Changes Between Modes
5. Draw Your Watch Face

You should also read

- Watch Faces for Android Wear

Related Samples

- WatchFace

After you have configured your project and added a class that implements the watch face service, you can start writing code to initialize and draw your custom watch face.

This lesson includes examples from the WatchFace sample to show how the system uses the watch face service. Many aspects of the service implementations described here (such as initialization and device features detection) apply to any watch face, so you can reuse some of the code in your own watch faces.

Figure 1. The analog and digital watch faces in the WatchFace sample.

Initialize Your Watch Face

When the system loads your service, you should allocate and initialize most of the resources that your watch face needs, including loading bitmap resources, creating timer objects to run custom animations, configuring paint styles, and performing other computations. You can usually perform these operations only once and reuse their results. This practice improves the performance of your watch face and makes it easier to maintain your code.

To initialize your watch face, follow these steps:

1. Declare variables for a custom timer, graphic objects, and other elements.
2. Initialize the watch face elements in the `Engine.onCreate()` method.
3. Initialize the custom timer in the `Engine.onVisibilityChanged()` method.

The following sections describe these steps in detail.

Declare variables

The resources that you intialize when the system loads your service need to be accessible at different points throughout your implementation, so you can reuse them. You achieve this by declaring member variables for these resources in your `WatchFaceService.Engine` implementation.

Declare variables for the following elements:

Graphic objects
: Most watch faces contain at least one bitmap image used as the background of the watch face, as described in Create an Implementation Strategy. You can use additional bitmap images that represent clock hands or other design elements of your watch face.

Periodic timer
: The system notifies the watch face once a minute when the time changes, but some watch faces run animations at custom time intervals. In these cases, you need to provide a custom timer that ticks with the frequency required to update your watch face.

Time zone change receiver
: Users can adjust their time zone when they travel, and the system broadcasts this event. Your service implementation must register a broadcast receiver that is notified when the time zone changes and update the time accordingly.

The following snippet shows how to define these variables:

```
private class Engine extends CanvasWatchFaceService.Engine {
    static final int MSG_UPDATE_TIME = 0;

    Calendar mCalendar;

    // device features
    boolean mLowBitAmbient;

    // graphic objects
    Bitmap mBackgroundBitmap;
    Bitmap mBackgroundScaledBitmap;
    Paint mHourPaint;
    Paint mMinutePaint;
    ...

    // handler to update the time once a second in interactive mode
    final Handler mUpdateTimeHandler = new Handler() {
        @Override
        public void handleMessage(Message message) {
            switch (message.what) {
                case MSG_UPDATE_TIME:
                    invalidate();
                    if (shouldTimerBeRunning()) {
                        long timeMs = System.currentTimeMillis();
                        long delayMs = INTERACTIVE_UPDATE_RATE_MS
                                - (timeMs %
INTERACTIVE_UPDATE_RATE_MS);
                        mUpdateTimeHandler
                            .sendEmptyMessageDelayed(MSG_UPDATE_TIME,
```

```
delayMs);
                    }
                    break;
            }
        }
    };

    // receiver to update the time zone
    final BroadcastReceiver mTimeZoneReceiver = new
BroadcastReceiver() {
        @Override
        public void onReceive(Context context, Intent intent) {
            mCalendar.setTimeZone(TimeZone.getDefault());
            invalidate();
        }
    };

    // service methods (see other sections)
    ...
}
```

In the example above, the custom timer is implemented as a `Handler` instance that sends and processes delayed messages using the thread's message queue. For this particular watch face, the custom timer ticks once every second. When the timer ticks, the handler calls the `invalidate()` method and the system then calls the `onDraw()` method to redraw the watch face.

Initialize watch face elements

After declaring member variables for bitmap resources, paint styles, and other elements that you reuse every time you redraw your watch face, initialize them when the system loads your service. Initializing these elements only once and reusing them improves performance and battery life.

In the `Engine.onCreate()` method, initialize the following elements:

- Load the background image.
- Create styles and colors to draw graphic objects.
- Allocate an object to calculate the time.
- Configure the system UI.

The following snippet shows how to initialize these elements:

```
@Override
public void onCreate(SurfaceHolder holder) {
```

```
    super.onCreate(holder);

    // configure the system UI (see next section)
    ...

    // load the background image
    Resources resources = AnalogWatchFaceService.this.getResources();
    Drawable backgroundDrawable =
resources.getDrawable(R.drawable.bg, null);
    mBackgroundBitmap = ((BitmapDrawable)
backgroundDrawable).getBitmap();

    // create graphic styles
    mHourPaint = new Paint();
    mHourPaint.setARGB(255, 200, 200, 200);
    mHourPaint.setStrokeWidth(5.0f);
    mHourPaint.setAntiAlias(true);
    mHourPaint.setStrokeCap(Paint.Cap.ROUND);
    ...

    // allocate a Calendar to calculate local time using the UTC
time and time zone
    mCalendar = Calendar.getInstance();
}
```

The background bitmap is loaded only once when the system initializes the watch face. The graphic styles are instances of the `Paint` class. Use these styles to draw the elements of your watch face inside the `Engine.onDraw()` method, as described in Drawing Your Watch Face.

Initialize the custom timer

As a watch face developer, you decide how often you want to update your watch face by providing a custom timer that ticks with the required frequency while the device is in interactive mode. This enables you to create custom animations and other visual effects.

Note: In ambient mode, the system does not reliably call the custom timer. To update the watch face in ambient mode, see Update the watch face in ambient mode.

An example timer definition from the `AnalogWatchFaceService` class that ticks once every second is shown in Declare variables. In the `Engine.onVisibilityChanged()` method, start the custom timer if these two conditions apply:

- The watch face is visible.

- The device is in interactive mode.

The `AnalogWatchFaceService` class schedules the next timer tick if required as follows:

```
private void updateTimer() {
    mUpdateTimeHandler.removeMessages(MSG_UPDATE_TIME);
    if (shouldTimerBeRunning()) {
        mUpdateTimeHandler.sendEmptyMessage(MSG_UPDATE_TIME);
    }
}

private boolean shouldTimerBeRunning() {
    return isVisible() && !isInAmbientMode();
}
```

This custom timer ticks once every second, as described in Declare variables.

In the `onVisibilityChanged()` method, start the timer if required and register the receiver for time zone changes as follows:

```
@Override
public void onVisibilityChanged(boolean visible) {
    super.onVisibilityChanged(visible);

    if (visible) {
        registerReceiver();

        // Update time zone in case it changed while we weren't
visible.
        mCalendar.setTimeZone(TimeZone.getDefault());
    } else {
        unregisterReceiver();
    }

    // Whether the timer should be running depends on whether we're
visible and
    // whether we're in ambient mode, so we may need to start or
stop the timer
    updateTimer();
}
```

When the watch face is visible, the `onVisibilityChanged()` method registers the receiver for time zone changes. If the device is in interactive mode, this method also starts the custom timer. When the watch face is not visible, this method stops the custom timer and unregisters the receiver for time zone changes. The `registerReceiver()` and `unregisterReceiver()` methods are implemented as follows:

```
private void registerReceiver() {
    if (mRegisteredTimeZoneReceiver) {
        return;
    }
    mRegisteredTimeZoneReceiver = true;
    IntentFilter filter = new
IntentFilter(Intent.ACTION_TIMEZONE_CHANGED);
    AnalogWatchFaceService.this.registerReceiver(mTimeZoneReceiver,
filter);
}

private void unregisterReceiver() {
    if (!mRegisteredTimeZoneReceiver) {
        return;
    }
    mRegisteredTimeZoneReceiver = false;

AnalogWatchFaceService.this.unregisterReceiver(mTimeZoneReceiver);
}
```

Update the watch face in ambient mode

In ambient mode, the system calls the `Engine.onTimeTick()` method every minute. It is usually sufficient to update your watch face once per minute in this mode. To update your watch face while in interactive mode, you must provide a custom timer as described in Initialize the custom timer.

In ambient mode, most watch face implementations simply invalidate the canvas to redraw the watch face in the `Engine.onTimeTick()` method:

```
@Override
public void onTimeTick() {
    super.onTimeTick();

    invalidate();
}
```

Configure the System UI

Watch faces should not interfere with system UI elements, as described in Accommodate System UI Elements. If your watch face has a light background or shows information near the bottom of the screen, you may have to configure the size of notification cards or enable background protection.

Android Wear enables you to configure the following aspects of the system UI when your watch face is active:

- Specify how far the first notification card peeks into the screen.
- Specify whether the system draws the time over your watch face.
- Show or hide cards when in ambient mode.
- Protect the system indicators with a solid background around them.
- Specify the positioning of the system indicators.

To configure these aspects of the system UI, create a `WatchFaceStyle` instance and pass it to the `Engine.setWatchFaceStyle()` method.

The `AnalogWatchFaceService` class configures the system UI as follows:

```
@Override
public void onCreate(SurfaceHolder holder) {
    super.onCreate(holder);

    // configure the system UI
    setWatchFaceStyle(new
WatchFaceStyle.Builder(AnalogWatchFaceService.this)
            .setCardPeekMode(WatchFaceStyle.PEEK_MODE_SHORT)
            .setBackgroundVisibility(WatchFaceStyle

.BACKGROUND_VISIBILITY_INTERRUPTIVE)
            .setShowSystemUiTime(false)
            .build());
    ...
}
```

The code snippet above configures peeking cards to be a single line tall, the background of a peeking card to show only briefly and only for interruptive notifications, and the system time not to be shown (since this watch face draws its own time representation).

You can configure the style of the system UI at any point in your watch face implementation. For example, if the user selects a white background, you can add background protection for the system indicators.

For more details about configuring the system UI, see the Wear API reference documentation.

Obtain Information About the Device Screen

The system calls the `Engine.onPropertiesChanged()` method when it

determines the properties of the device screen, such as whether the device uses low-bit ambient mode and whether the screen requires burn-in protection.

The following code snippet shows how to obtain these properties:

```
@Override
public void onPropertiesChanged(Bundle properties) {
    super.onPropertiesChanged(properties);
    mLowBitAmbient = properties.getBoolean(PROPERTY_LOW_BIT_AMBIENT,
false);
    mBurnInProtection =
properties.getBoolean(PROPERTY_BURN_IN_PROTECTION,
            false);
}
```

You should take these device properties into account when drawing your watch face:

- For devices that use low-bit ambient mode, the screen supports fewer bits for each color in ambient mode, so you should disable anti-aliasing and bitmap filtering when the device switches to ambient mode.
- For devices that require burn-in protection, avoid using large blocks of white pixels in ambient mode and do not place content within 10 pixels of the edge of the screen, since the system shifts the content periodically to avoid pixel burn-in.

For more information about low-bit ambient mode and burn-in protection, see Optimize for Special Screens. For more information on how to disable bitmap filtering, see Bitmap Filtering.

Respond to Changes Between Modes

When the device switches between ambient and interactive modes, the system calls the `Engine.onAmbientModeChanged()` method. Your service implementation should make any necessary adjustments to switch between modes and then call the `invalidate()` method for the system to redraw the watch face.

The following snippet shows how to implement this method:

```
@Override
public void onAmbientModeChanged(boolean inAmbientMode) {

    super.onAmbientModeChanged(inAmbientMode);

    if (mLowBitAmbient) {
        boolean antiAlias = !inAmbientMode;
        mHourPaint.setAntiAlias(antiAlias);
        mMinutePaint.setAntiAlias(antiAlias);
        mSecondPaint.setAntiAlias(antiAlias);
        mTickPaint.setAntiAlias(antiAlias);
    }
    invalidate();
    updateTimer();
}
```

This example makes adjustments to some graphic styles and invalidates the canvas so the system can redraw the watch face.

Draw Your Watch Face

To draw a custom watch face, the system calls the `Engine.onDraw()` method with a `Canvas` instance and the bounds in which you should draw your watch face. The bounds take into account any inset areas, such as the "chin" on the bottom of some round devices. You can use this canvas to draw your watch face directly as follows:

1. Override the `onSurfaceChanged()` method to scale your background to fit the device any time the view changes.

   ```
   @Override
   public void onSurfaceChanged(
           SurfaceHolder holder, int format, int width, int height)
   {
       if (mBackgroundScaledBitmap == null
               || mBackgroundScaledBitmap.getWidth() != width
               || mBackgroundScaledBitmap.getHeight() != height) {
           mBackgroundScaledBitmap =
   Bitmap.createScaledBitmap(mBackgroundBitmap,
                   width, height, true /* filter */);
       }
       super.onSurfaceChanged(holder, format, width, height);
   }
   ```

2. Check whether the device is in ambient mode or interactive mode.
3. Perform any required graphic computations.
4. Draw your background bitmap on the canvas.
5. Use the methods in the `Canvas` class to draw your watch face.

The following snippet shows how to implement the `onDraw()` method:

```
@Override
public void onDraw(Canvas canvas, Rect bounds) {
    // Update the time
    mCalendar.setTimeInMillis(System.currentTimeMillis());

    // Constant to help calculate clock hand rotations
    final float TWO_PI = (float) Math.PI * 2f;

    int width = bounds.width();
    int height = bounds.height();

    canvas.drawBitmap(mBackgroundScaledBitmap, 0, 0, null);

    // Find the center. Ignore the window insets so that, on round
watches
    // with a "chin", the watch face is centered on the entire
screen, not
    // just the usable portion.
    float centerX = width / 2f;
    float centerY = height / 2f;

    // Compute rotations and lengths for the clock hands.
    float seconds = mCalendar.get(Calendar.SECOND) +
                    mCalendar.get(Calendar.MILLISECOND) / 1000f;
    float secRot = seconds / 60f * TWO_PI;
    float minutes = mCalendar.get(Calendar.MINUTE) + seconds / 60f;
    float minRot = minutes / 60f * TWO_PI;
    float hours = mCalendar.get(Calendar.HOUR) + minutes / 60f;
    float hrRot = hours / 12f * TWO_PI;

    float secLength = centerX - 20;
    float minLength = centerX - 40;
    float hrLength = centerX - 80;

    // Only draw the second hand in interactive mode.
    if (!isInAmbientMode()) {
        float secX = (float) Math.sin(secRot) * secLength;
        float secY = (float) -Math.cos(secRot) * secLength;
        canvas.drawLine(centerX, centerY, centerX + secX, centerY +
                        secY, mSecondPaint);
    }

    // Draw the minute and hour hands.
    float minX = (float) Math.sin(minRot) * minLength;
    float minY = (float) -Math.cos(minRot) * minLength;
    canvas.drawLine(centerX, centerY, centerX + minX, centerY +
minY,
                    mMinutePaint);
    float hrX = (float) Math.sin(hrRot) * hrLength;
    float hrY = (float) -Math.cos(hrRot) * hrLength;
    canvas.drawLine(centerX, centerY, centerX + hrX, centerY + hrY,
                    mHourPaint);
}
```

This method computes the required positions for the clock hands based on the current time and draws them on top of the background bitmap

using the graphic styles initialized in the `onCreate()` method. The second hand is only drawn in interactive mode, not in ambient mode.

For more information about drawing on a Canvas instance, see Canvas and Drawables.

The WatchFace sample includes additional watch faces that you can refer to as examples of how to implement the `onDraw()` method.

Next Get started Next class

Showing Information in Watch Faces

Previous Next Get started

This lesson teaches you to

1. Create a Compelling Experience
2. Add Data to Your Watch Face

Related Samples

- WatchFace

You should also read

- Watch Faces for Android Wear

In addition to telling time, Android Wear devices provide users with contextually relevant information in the form of cards, notifications, and other wearable apps. Creating a custom watch face not only gives you the opportunity to tell time in visually compelling ways, but also to show users relevant information whenever they glance at their device.

Like any other wearable app, your watch face can communicate with apps running on the handheld device using the Wearable Data Layer API. In some cases, you need to create an activity in the handheld app module of your project that retrieves data from the Internet or from the user's profile and then shares it with your watch face.

Figure 1. Examples of watch faces with integrated data.

Create a Compelling Experience

Before you design and implement a contextually-aware watch face, answer the following questions:

- What kind of data do you want to incorporate?
- Where can you obtain this data?
- How often does the data change significantly?
- How can you present the data such that users understand it at a glance?

Android Wear devices are usually paired with a companion device that has a GPS sensor and cellular connectivity, so you have endless possibilities to integrate different kinds of data in your watch face, such as location, calendar events, social media trends, picture feeds, stock market quotes, news events, sports scores, and so on. However, not all kinds of data are appropriate for a watch face, so you should consider what kinds of data are most relevant to your users throughout the day. Your watch face should also gracefully handle the case where the wearable is not paired with a companion device or when an Internet connection is not available.

The active watch face on an Android Wear device is an app that runs continuously, so you must retrieve data in a battery-efficient manner. For example, you can obtain the current weather every ten minutes and store the results locally, instead of requesting an update every minute. You can

also refresh contextual data when the device switches from ambient to interactive mode, since the user is more likely to glance at the watch when this transition occurs.

You should summarize contextual information on your watch face, since there is limited space available on the screen and users just glance at their watch for a second or two at a time. Sometimes the best way to convey contextual information is to react to it using graphics and colors. For example, a watch face could change its background image depending on the current weather.

Add Data to Your Watch Face

The WatchFace sample demonstrates how to obtain calendar data from the user's profile in the `CalendarWatchFaceService` class and shows how many meetings there are in the following twenty-four hours.

Figure 2. The calendar watch face.

To implement a watch face that incorporates contextual data, follow these steps:

1. Provide a task that retrieves the data.
2. Create a custom timer to invoke your task periodically, or notify your watch face service when external data changes.
3. Redraw your watch face with the updated data.

The following sections describe these steps in detail.

Provide a task to retrieve data

Create a class inside your `CanvasWatchFaceService.Engine` implementation that extends `AsyncTask` and add the code to retrieve the data you're

interested in.

The `CalendarWatchFaceService` class obtains the number of meetings in the next day as follows:

```
/* Asynchronous task to load the meetings from the content provider
and
 * report the number of meetings back using onMeetingsLoaded() */
private class LoadMeetingsTask extends AsyncTask<Void, Void,
Integer> {
    @Override
    protected Integer doInBackground(Void... voids) {
        long begin = System.currentTimeMillis();
        Uri.Builder builder =

WearableCalendarContract.Instances.CONTENT_URI.buildUpon();
        ContentUris.appendId(builder, begin);
        ContentUris.appendId(builder, begin +
DateUtils.DAY_IN_MILLIS);
        final Cursor cursor = getContentResolver()
.query(builder.build(),
                null, null, null, null);
        int numMeetings = cursor.getCount();
        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            Log.v(TAG, "Num meetings: " + numMeetings);
        }
        return numMeetings;
    }

    @Override
    protected void onPostExecute(Integer result) {
        /* get the number of meetings and set the next timer tick */
        onMeetingsLoaded(result);
    }
}
```

The `WearableCalendarContract` class from the Wearable Support Library provides direct access to the user's calendar events from the companion device.

When the task finishes retrieving data, your code invokes a callback method. The following sections describe how to implement the callback method in detail.

For more information about obtaining data from the calendar, see the Calendar Provider API guide.

Create a custom timer

You can implement a custom timer that ticks periodically to update your

data. The `CalendarWatchFaceService` class uses a `Handler` instance that sends and processes delayed messages using the thread's message queue:

```
private class Engine extends CanvasWatchFaceService.Engine {
    ...
    int mNumMeetings;
    private AsyncTask<Void, Void, Integer> mLoadMeetingsTask;

    /* Handler to load the meetings once a minute in interactive
mode. */
    final Handler mLoadMeetingsHandler = new Handler() {
        @Override
        public void handleMessage(Message message) {
            switch (message.what) {
                case MSG_LOAD_MEETINGS:
                    cancelLoadMeetingTask();
                    mLoadMeetingsTask = new LoadMeetingsTask();
                    mLoadMeetingsTask.execute();
                    break;
            }
        }
    };
    ...
}
```

This method initializes the timer when the watch face becomes visible:

```
@Override
public void onVisibilityChanged(boolean visible) {
    super.onVisibilityChanged(visible);
    if (visible) {
        mLoadMeetingsHandler.sendEmptyMessage(MSG_LOAD_MEETINGS);
    } else {
        mLoadMeetingsHandler.removeMessages(MSG_LOAD_MEETINGS);
        cancelLoadMeetingTask();
    }
}
```

The next timer tick is set in the `onMeetingsLoaded()` method, as shown in the next section.

Redraw your watch face with the updated data

When the task that retrieves your data finishes, call the `invalidate()` method so the system redraws your watch face. Store your data inside member variables of the `Engine` class so you can access it inside the `onDraw()` method.

The `CalendarWatchFaceService` class provides a callback method for the

task to invoke when it finishes retrieving calendar data:

```
private void onMeetingsLoaded(Integer result) {
    if (result != null) {
        mNumMeetings = result;
        invalidate();
    }
    if (isVisible()) {
        mLoadMeetingsHandler.sendEmptyMessageDelayed(
                MSG_LOAD_MEETINGS, LOAD_MEETINGS_DELAY_MS);
    }
}
```

The callback method stores the result in a member variable, invalidates the view, and schedules the next timer tick to run the task again.

Next Get started Next class

Providing Configuration Activities

Previous Next Get started

This lesson teaches you to

1. Specify an Intent for Configuration Activities
2. Create a Wearable Configuration Activity
3. Create a Companion Configuration Activity

Related Samples

- WatchFace

You should also read

- Watch Faces for Android Wear

When users install a handheld app that contains a wearable app with watch faces, these watch faces become available in the Android Wear companion app on the companion device and in the watch face picker on the wearable. Users can choose the active watch face for their wearable device by selecting it on the companion app or using the watch face picker on the wearable device.

Some watch faces support configuration parameters to let users customize how the watch face looks and behaves. For example, some watch faces let users pick a custom background color, and watch faces that tell time for two different time zones can let users select which time zones they are interested in.

Watch faces that support configuration parameters can let users customize a watch face using an activity in the wearable app, an activity on the handheld app, or both. Users can start the wearable configuration activity on the wearable device, and they can start the companion configuration activity from the Android Wear companion app.

The digital watch face from the WatchFace sample demonstrates how to implement handheld and wearable configuration activities and how to update a watch face in response to configuration changes.

Specify an Intent for Configuration Activities

If your watch face includes configuration activities, add the following metadata entries to the service declaration in the manifest file of the wearable app:

```
<service
    android:name=".DigitalWatchFaceService" ... />
    <!-- companion configuration activity -->
    <meta-data
        android:name=

"com.google.android.wearable.watchface.companionConfigurationAction"
        android:value=
           "com.example.android.wearable.watchface.CONFIG_DIGITAL" />
    <!-- wearable configuration activity -->
    <meta-data
        android:name=

"com.google.android.wearable.watchface.wearableConfigurationAction"
        android:value=
           "com.example.android.wearable.watchface.CONFIG_DIGITAL" />
    ...
</service>
```

Provide values for these entries that are preceded by the package name of your app. Configuration activities register intent filters for this intent, and the system fires this intent when users want to configure your watch face.

If your watch face only includes a companion or a wearable configuration activity, you only need to include the corresponding metadata entry from the example above.

Create a Wearable Configuration Activity

Wearable configuration activities provide a limited set of customization choices for a watch face, because complex menus are hard to navigate on smaller screens. Your wearable configuration activity should provide binary choices and just a few selections to customize the main aspects of your watch face.

To create a wearable configuration activity, add a new activity to your wearable app module and declare the following intent filter in the manifest file of the wearable app:

```
<activity
    android:name=".DigitalWatchFaceWearableConfigActivity"
    android:label="@string/digital_config_name">
    <intent-filter>
        <action android:name=
            "com.example.android.wearable.watchface.CONFIG_DIGITAL"
/>
        <category android:name=

"com.google.android.wearable.watchface.category.WEARABLE_CONFIGURATION'
/>
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
```

The name of the action in this intent filter must match the intent name you defined in Specify an Intent for Configuration Activities.

In your configuration activity, build a simple UI that provides selections for users to customize your watch face. When users make a selection, use the Wearable Data Layer API to communicate the configuration change to the watch face activity.

For more details, see the `DigitalWatchFaceWearableConfigActivity` and `DigitalWatchFaceUtil` classes in the WatchFace sample.

Create a Companion Configuration Activity

Companion configuration activities give users access to the full set of configuration choices for a watch face, because it is easier to interact

with complex menus on the larger screen of a handheld device. For example, a configuration activity on a handheld device enables you to present users with elaborate color pickers to select the background color of a watch face.

To create a companion configuration activity, add a new activity to your handheld app module and declare the following intent filter in the manifest file of the handheld app:

```
<activity
    android:name=".DigitalWatchFaceCompanionConfigActivity"
    android:label="@string/app_name">
    <intent-filter>
        <action android:name=
            "com.example.android.wearable.watchface.CONFIG_DIGITAL"
/>
        <category android:name=

"com.google.android.wearable.watchface.category.COMPANION_CONFIGURATION
/>
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
```

In your configuration activity, build a UI that provides options to customize all the configurable elements of your watch face. When users make a selection, use the Wearable Data Layer API to communicate the configuration change to the watch face activity.

For more details, see the `DigitalWatchFaceCompanionConfigActivity` class in the WatchFace sample.

Create a Listener Service in the Wearable App

To receive updated configuration parameters from the configuration activities, create a service that implements the `WearableListenerService` interface from the Wearable Data Layer API in your wearable app. Your watch face implementation can redraw the watch face when the configuration parameters change.

For more details, see the `DigitalWatchFaceConfigListenerService` and `DigitalWatchFaceService` classes in the WatchFace sample.

Next Get started Next class

Addressing Common Issues

Previous Next Get started

This lesson teaches you to

1. Detect the Shape of the Screen
2. Accomodate Peek Cards
3. Use Relative Measurements

You should also read

- Watch Faces for Android Wear

Creating a custom watch face for Android Wear is substantially different from creating notifications and wearable-specific activities. This class shows you how to resolve some issues that you may encounter as you implement your first few watch faces.

Detect the Shape of the Screen

Some Android Wear devices have square screens, while others have round screens. Devices with round screens can contain an inset (or "chin") at the bottom of the screen. Your watch face should adapt to and take advantage of the particular shape of the screen, as described in the design guidelines.

Android Wear lets your watch face determine the screen shape at runtime. To detect whether the screen is square or round, override the `onApplyWindowInsets()` method in the `CanvasWatchFaceService.Engine` class as follows:

```
private class Engine extends CanvasWatchFaceService.Engine {
    boolean mIsRound;
```

```
    int mChinSize;

    @Override
    public void onApplyWindowInsets(WindowInsets insets) {
        super.onApplyWindowInsets(insets);
        mIsRound = insets.isRound();
        mChinSize = insets.getSystemWindowInsetBottom();
    }
    ...
}
```

To adapt your design when you draw your watch face, check the value of the `mIsRound` and `mChinSize` member variables.

Accomodate Peek Cards

When users receive a notification, the notification card may cover a significant portion of the screen, depending on the system UI style. Your watch face should adapt to these situations by ensuring that users can still tell the time while the notification card is present.

Figure 1. Some analog watch faces require adjustments when notification cards appear.

Analog watch faces can make adjustments when a notification card is present, like scaling down the watch face to fit inside the portion of the screen not covered by the peek card. Digital watch faces that display the time in the area of the screen not covered by peek cards do not usually require adjustments. To determine the free space above the peek card so you can adapt your watch face, use the `WatchFaceService.Engine.getPeekCardPosition()` method.

In ambient mode, peek cards have a transparent background. If your watch face contains details near the card in ambient mode, consider drawing a black rectangle over them to ensure that users can read the contents of the card.

Configure the System Indicators

To ensure that the system indicators remain visible, you can configure their position on the screen and whether they need background protection when you create a `WatchFaceStyle` instance:

Figure 2. The status bar.

- To set the position of the status bar, use the `setStatusBarGravity()` method.
- To set the position of the hotword, use the `setHotwordIndicatorGravity()` method.
- To protect the status bar and hotword with a semi-transparent gray background, use the `setViewProtection()` method. This is usually necessary if your watch face has a light background, since the system indicators are white.

For more information about the system indicators, see Configure the System UI and read the design guidelines.

Use Relative Measurements

Android Wear devices from different manufacturers feature screens with a variety of sizes and resolutions. Your watch face should adapt to these variations by using relative measurements instead of absolute pixel values.

When you draw your watch face, obtain the size of the canvas with the `Canvas.getWidth()` and `Canvas.getHeight()` methods and set the positions of your graphic elements using values that are some fraction of the detected screen size. If you resize the elements of your watch face in response to a peek card, use values that are some fraction of the remaining space above the card to redraw your watch face.

Next Get started Next class

Optimizing Performance and Battery Life

Previous Next Get started

This lesson teaches you to

1. Reduce the Size of Your Bitmap Assets
2. Combine Bitmap Assets
3. Disable Anti-Aliasing when Drawing Scaled Bitmaps
4. Move Expensive Operations Outside the Drawing Method
5. Follow Best Practices to Save Power

You should also read

- Watch Faces for Android Wear

In addition to accommodating notification cards and system indicators, you need to ensure that the animations in your watch face run smoothly and that your service does not perform unnecessary computations. Watch faces in Android Wear run continuously on the device, so it is critical that your watch face uses power efficiently.

This lesson provides some tips to speed up your animations and to measure and conserve power on the device.

Reduce the Size of Your Bitmap Assets

Many watch faces consist of a background image and other graphic assets that are transformed and overlapped on top of the background image, such as clock hands and other elements of the design that move over time. Typically these graphic elements are rotated (and sometimes

scaled) inside the `Engine.onDraw()` method every time the system redraws the watch face, as described in Draw Your Watch Face.

The larger these graphic assets are, the more computationally expensive it is to transform them. Transforming large graphic assets in the `Engine.onDraw()` method drastically reduces the frame rate at which the system can run your animations.

To improve the performance of your watch face:

- Do not use graphic elements that are larger than you need.
- Remove extra transparent pixels around the edges.

The example clock hand on the left side of Figure 1 can be reduced in size by 97%.

Figure 1. Clock hands can be cropped to remove extra pixels.

Reducing the size of your bitmap assets as described in this section not only improves the performance of your animations, but it also saves power.

Combine Bitmap Assets

If you have bitmaps that are often drawn together, consider combining them into the same graphic asset. You can often combine the background image in interactive mode with the tick marks to avoid drawing two full-screen bitmaps every time the system redraws the watch face.

Disable Anti-Aliasing when Drawing Scaled Bitmaps

When you draw a scaled bitmap on the `Canvas` object using the `Canvas.drawBitmap()` method, you can provide a `Paint` instance to

configure several options. To improve performance, disable anti-aliasing using the `setAntiAlias()` method, since this option does not have any effect on bitmaps.

Use bitmap filtering

For bitmap assets that you draw on top of other elements, enable bitmap filtering on the same `Paint` instance using the `setFilterBitmap()` method. Figure 2 shows a magnified view of a clock hand with and without bitmap filtering.

Figure 2. Example of bitmap filtering disabled (left) and enabled (right).

Note: In low-bit ambient mode, the system does not reliably render the colors in the image for bitmap filtering to process successfully. When ambient mode is active, disable bitmap filtering.

Move Expensive Operations Outside the Drawing Method

The system calls the `Engine.onDraw()` method every time it redraws your watch face, so you should only include operations that are strictly required to update the watch face inside this method to improve performance.

When possible, avoid performing these operations inside the `Engine.onDraw()` method:

- Loading images and other resources.
- Resizing images.
- Allocating objects.
- Performing computations whose result does not change between frames.

You can usually perform these operations in the `Engine.onCreate()` method instead. You can resize images ahead of time in the

`Engine.onSurfaceChanged()` method, which provides you with the size of the canvas.

To analyze the performance of your watch face, use the Android Device Monitor. In particular, ensure that the execution time for your `Engine.onDraw()` implementation is short and consistent across invocations. For more information, see Using DDMS.

Follow Best Practices to Save Power

In addition to the techniques described in the previous sections, follow the best practices in this section to reduce the power consumption of your watch face.

Reduce the frame rate of animations

Animations are often computationally expensive and consume a significant amount of power. Most animations look fluid at 30 frames per second, so you should avoid running your animations at a higher frame rate.

Let the CPU sleep

Animations and small changes to the contents of the watch face wake up the CPU. Your watch face should let the CPU sleep in between animations. For example, you can use short bursts of animation every second in interactive mode and then let the CPU sleep until the next second. Letting the CPU sleep often, even briefly, can significantly reduce power consumption.

To maximize battery life, use animations sparingly. Even a blinking colon wakes up the CPU with every blink and hurts battery life.

Monitor power consumption

The Android Wear companion app lets developers and users see how much battery different processes on the wearable device are consuming

under **Settings** > **Watch battery**.

For more information about new features in Android 5.0 that help you improve battery life, see Project Volta.

Next Get started Next class

Detecting Location on Android Wear

In this document

1. Connect to Google Play Services
2. Request Location Updates
3. Detect On-Board GPS
4. Handle Disconnection Events
5. Handle Location Not Found
6. Synchronize Data

Dependencies and prerequisites

- Android 4.3 (API Level 18) or higher on the handset device
- Google Play services 6.1 or higher
- An Android Wear device

See also

- Making Your App Location-Aware

Location awareness on wearable devices enables you to create apps that give users a better understanding of their geographic position, movement and what's around them. With the small form factor and glanceable nature of a wearable device, you can build low-friction apps that record and respond to location data.

Some wearable devices include a GPS sensor that can retrieve location data without another tethered device. However, when you request location data in a wearable app, you don't have to worry about where the location data originates; the system retrieves the location updates using

the most power-efficient method. Your app should be able to handle loss of location data, in case the wear device loses connection with its paired device and does not have a built-in GPS sensor.

This document shows you how to check for on-device location sensors, receive location data, and monitor tethered data connections.

Note: The article assumes that you know how to use the Google Play services API to retrieve location data. For more information, see Making Your App Location-Aware.

Connect to Google Play Services

Location data on wearable devices is obtained though the Google Play services location APIs. You use the `FusedLocationProviderApi` and its accompanying classes to obtain this data. To access location services, create an instance of `GoogleApiClient`, which is the main entry point for any of the Google Play services APIs.

Caution: Do not use the existing Location APIs in the Android framework. The best practice for retrieving location updates is through the Google Play services API as outlined in this article.

To connect to Google Play services, configure your app to create an instance of `GoogleApiClient`:

1. Create an activity that specifies an implementation for the interfaces `ConnectionCallbacks`, `OnConnectionFailedListener`, and `LocationListener`.
2. In your activity's `onCreate()` method, create an instance of `GoogleApiClient` and add the Location service.
3. To gracefully manage the lifecycle of the connection, call `connect()` in the `onResume()` method and `disconnect()` in the `onPause()` method.

The following code example shows an implementation of an activity that implements the `LocationListener` interface:

```
public class WearableMainActivity extends Activity implements
    GoogleApiClient.ConnectionCallbacks,
```

```
    GoogleApiClient.OnConnectionFailedListener,
    LocationListener {

    private GoogleApiClient mGoogleApiClient;
    ...

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        ...
        mGoogleApiClient = new GoogleApiClient.Builder(this)
                .addApi(LocationServices.API)
                .addApi(Wearable.API)  // used for data layer API
                .addConnectionCallbacks(this)
                .addOnConnectionFailedListener(this)
                .build();
    }

    @Override
    protected void onResume() {
        super.onResume();
        mGoogleApiClient.connect();
        ...
    }

    @Override
    protected void onPause() {
        super.onPause();
        ...
        mGoogleApiClient.disconnect();
    }
}
```

For more information on connecting to Google Play services, see Accessing Google APIs.

Request Location Updates

After your app has connected to the Google Play services API, it is ready to start receiving location updates. When the system invokes the `onConnected()` callback for your client, you build the location data request as follows:

1. Create a `LocationRequest` object and set any options using methods like `setPriority()`.
2. Request location updates using `requestLocationUpdates()`.
3. Remove location updates using `removeLocationUpdates()` in the `onPause()` method.

The following example shows how to retrieve and remove location updates:

```
@Override
public void onConnected(Bundle bundle) {
    LocationRequest locationRequest = LocationRequest.create()
            .setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY)
            .setInterval(UPDATE_INTERVAL_MS)
            .setFastestInterval(FASTEST_INTERVAL_MS);

    LocationServices.FusedLocationApi
            .requestLocationUpdates(mGoogleApiClient,
locationRequest, this)
            .setResultCallback(new ResultCallback() {

                @Override
                public void onResult(Status status) {
                    if (status.getStatus().isSuccess()) {
                        if (Log.isLoggable(TAG, Log.DEBUG)) {
                            Log.d(TAG, "Successfully requested
location updates");
                        }
                    } else {
                        Log.e(TAG,
                                "Failed in requesting location
updates, "
                                        + "status code: "
                                        + status.getStatusCode()
                                        + ", message: "
                                        + status.getStatusMessage());
                    }
                }
            });
}

@Override
protected void onPause() {
    super.onPause();
    if (mGoogleApiClient.isConnected()) {
        LocationServices.FusedLocationApi
             .removeLocationUpdates(mGoogleApiClient, this);
    }
    mGoogleApiClient.disconnect();
}

@Override
public void onConnectionSuspended(int i) {
    if (Log.isLoggable(TAG, Log.DEBUG)) {
        Log.d(TAG, "connection to location client suspended");
    }
}
```

Now that you have enabled location updates, the system calls the `onLocationChanged()` method with the updated location at the interval specified in `setInterval()`

Detect On-Board GPS

Not all wearables have a GPS sensor. If your user goes out for a run and leaves their phone at home, your wearable app cannot receive location data through a tethered connection. If the wearable device does not have a sensor, you should detect this situation and warn the user that location functionality is not available.

To determine whether your Android Wear device has a built-in GPS sensor, use the `hasSystemFeature()` method. The following code detects whether the device has built-in GPS when you start an activity:

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setContentView(R.layout.main_activity);
    if (!hasGps()) {
        Log.d(TAG, "This hardware doesn't have GPS.");
        // Fall back to functionality that does not use location or
        // warn the user that location function is not available.
    }

    ...
}

private boolean hasGps() {
    return
getPackageManager().hasSystemFeature(PackageManager.FEATURE_LOCATION_GP

}
```

Handle Disconnection Events

Wearable devices relying on a tethered connection for location data may lose their connections abruptly. If your wearable app expects a constant stream of data, you must handle the disconnection based upon where that data is interrupted or unavailable. On a wearable device with no onboard GPS sensor, loss of location data occurs when the device loses its tethered data connection.

In cases where your app depends on a tethered data connection for location data and the wear device does not have a GPS sensor, you should detect the loss of that connection, warn the user, and gracefully

degrade the functionality of your app.

To detect the loss of a tethered data connection:

1. Extend a `WearableListenerService` that lets you listen for important data layer events.
2. Declare an intent filter in your Android manifest to notify the system about your `WearableListenerService`. This filter allows the system to bind your service as needed.

```
<service android:name=".NodeListenerService">
    <intent-filter>
        <action
android:name="com.google.android.gms.wearable.BIND_LISTENER" />
    </intent-filter>
</service>
```

3. Implement the `onPeerDisconnected()` method and handle cases of whether or not the device has built-in GPS.

```
public class NodeListenerService extends WearableListenerService
{

    private static final String TAG = "NodeListenerService";

    @Override
    public void onPeerDisconnected(Node peer) {
        Log.d(TAG, "You have been disconnected.");
        if(!hasGPS()) {
            // Notify user to bring tethered handset
            // Fall back to functionality that does not use
location
        }
    }
    ...
}
```

For more information, read the Listen for Data Layer Events guide.

Handle Location Not Found

When the GPS signal is lost, you can still retrieve the last known location using `getLastLocation()`. This method can be helpful in situations where you are unable to get a GPS fix, or when your wearable doesn't have built-in GPS and loses its connection with the phone.

The following code uses `getLastLocation()` to retrieve the last known

location if available:

```
Location location = LocationServices.FusedLocationApi
                .getLastLocation(mGoogleApiClient);
```

Synchronize Data

If your wearable app records data using the built-in GPS, you may want to synchronize the location data with the handset. With the `LocationListener`, you implement the `onLocationChanged()` method to detect and record the location as it changes.

The following code for wearable apps detects when the location changes and uses the data layer API to store the data for later retrieval by your phone app:

```
@Override
public void onLocationChanged(Location location) {
    ...
    addLocationEntry(location.getLatitude(),
location.getLongitude());

}

private void addLocationEntry(double latitude, double longitude) {
    if (!mSaveGpsLocation || !mGoogleApiClient.isConnected()) {
        return;
    }

    mCalendar.setTimeInMillis(System.currentTimeMillis());

    // Set the path of the data map
    String path = Constants.PATH + "/" +
mCalendar.getTimeInMillis();
    PutDataMapRequest putDataMapRequest =
PutDataMapRequest.create(path);

    // Set the location values in the data map
    putDataMapRequest.getDataMap()
            .putDouble(Constants.KEY_LATITUDE, latitude);
    putDataMapRequest.getDataMap()
            .putDouble(Constants.KEY_LONGITUDE, longitude);
    putDataMapRequest.getDataMap()
            .putLong(Constants.KEY_TIME,
mCalendar.getTimeInMillis());

    // Prepare the data map for the request
    PutDataRequest request = putDataMapRequest.asPutDataRequest();

    // Request the system to create the data item
    Wearable.DataApi.putDataItem(mGoogleApiClient, request)
            .setResultCallback(new ResultCallback() {
```

```
                @Override
                public void onResult(DataApi.DataItemResult
dataItemResult) {
                    if (!dataItemResult.getStatus().isSuccess()) {
                        Log.e(TAG, "Failed to set the data, "
                                + "status: " +
dataItemResult.getStatus()
                                .getStatusCode());
                    }
                }
            });
}
```

For more information on how to use the Data Layer API, see the Sending and Syncing Data guide.

Next Get started Next class

Building Apps for TV

These classes teach you how to build apps for TV devices.

Note: For details on how to publish your TV apps in Google Play, see Distribute to Android TV.

Building TV Apps

Previous Next Get started

Dependencies and Prerequisites

- Android 5.0 (API level 21) or higher

Android offers a rich user experience that's optimized for apps running on large screen devices, such as high-definition televisions. Apps on TV offer new opportunities to delight your users from the comfort of their couch.

TV apps use the same structure as those for phones and tablets. This approach means you can create new TV apps based on what you already know about building apps for Android, or extend your existing apps to also run on TV devices. However, the user interaction model for TV is substantially different from phone and tablet devices. In order to make your app successful on TV devices, you must design new layouts that can be easily understood from 10 feet away, and provide navigation that works with just a directional pad and a select button.

This class describes how to start building apps for TV, including setting up your development environment, basic requirements for layouts and navigation, as well as guidance on how to handle hardware features that are not typically available on TV devices.

Note: You are encouraged to use Android Studio for building TV apps, because it provides project setup, library inclusion, and packaging conveniences. This training assumes you are using Android Studio.

Lessons

Getting Started with TV Apps

Learn how to create a new Android Studio project for TV apps or

modify your existing app project to run on TV devices.

Building TV Layouts

Learn the minimum requirements for TV layouts and how to implement them.

Creating TV Navigation

Learn the requirements for TV navigation and how to implement TV-compatible navigation.

Handling TV Hardware

Learn how to check if your app is running on TV hardware, handle unsupported hardware features, and manage controller devices.

Next Get started Next class

Get Started with TV Apps

Previous Next Get started

This lesson teaches you how to

1. Determine Media Format Support
2. Setup a TV Project
3. Add TV Support Libraries
4. Build TV Apps
5. Run TV Apps

You should also read

1. TV Design
2. Building TV Layouts

TV apps use the same structure as those for phones and tablets. This similarity means you can modify your existing apps to also run on TV devices or create new apps based on what you already know about building apps for Android.

Important: There are specific requirements your app must meet to qualify as an Android TV app on Google Play. For more information, see the requirements listed in TV App Quality.

This lesson describes how to prepare your development environment for building TV apps, and the minimum required changes to enable an app to run on TV devices.

Determine Media Format Support

See the following documentation for information about the codecs,

protocols, and formats supported by Android TV.

- Supported Media Formats
- DRM
- `android.drm`
- ExoPlayer
- `android.media.MediaPlayer`

Set up a TV Project

This section discusses how to modify an existing app to run on TV devices, or create a new one. These are the main components you must use to create an app that runs on TV devices:

- **Activity for TV** (Required) - In your application manifest, declare an activity that is intended to run on TV devices.
- **TV Support Libraries** (Optional) - There are several Support Libraries available for TV devices that provide widgets for building user interfaces.

Prerequisites

Before you begin building apps for TV, you must:

- **Update your SDK tools to version 24.0.0 or higher**
  The updated SDK tools enable you to build and test apps for TV.
- **Update your SDK with Android 5.0 (API 21) or higher**
  The updated platform version provides new APIs for TV apps.
- **Create or update your app project**
  In order to access new APIs for TV devices, you must create a project or modify an existing project that targets Android 5.0 (API level 21) or higher.

Declare a TV Activity

An application intended to run on TV devices must declare a launcher activity for TV in its manifest using a `CATEGORY_LEANBACK_LAUNCHER` intent

filter. This filter identifies your app as being enabled for TV, and is required for your app to be considered a TV app in Google Play. Declaring this intent also identifies which activity in your app to launch when a user selects its icon on the TV home screen.

The following code snippet shows how to include this intent filter in your manifest:

```
<application
  android:banner="@drawable/banner" >
  ...
  <activity
    android:name="com.example.android.MainActivity"
    android:label="@string/app_name" >

    <intent-filter>
      <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
  </activity>

  <activity
    android:name="com.example.android.TvActivity"
    android:label="@string/app_name"
    android:theme="@style/Theme.Leanback">

    <intent-filter>
      <action android:name="android.intent.action.MAIN" />
      <category
android:name="android.intent.category.LEANBACK_LAUNCHER" />
    </intent-filter>

  </activity>
</application>
```

The second activity manifest entry in this example specifies that activity as the one to launch on a TV device.

Caution: If you do not include the `CATEGORY_LEANBACK_LAUNCHER` intent filter in your app, it is not visible to users running the Google Play store on TV devices. Also, if your app does not have this filter when you load it onto a TV device using developer tools, the app does not appear in the TV user interface.

If you are modifying an existing app for use on TV, your app should not use the same activity layout for TV that it does for phones and tablets. The user interface of your TV app (or TV portion of your existing app) should provide a simpler interface that can be easily navigated using a

remote control from a couch. For guidelines on designing an app for TV, see the TV Design guide. For more information on the minimum implementation requirements for interface layouts on TV, see Building TV Layouts.

Declare Leanback support

Declare that your app uses the Leanback user interface required by Android TV. If you are developing an app that runs on mobile (phones, wearables, tablets, etc.) as well as Android TV, set the `required` attribute value to `false`. If you set the `required` attribute value to `true`, your app will run only on devices that use the Leanback UI.

```
<manifest>
    <uses-feature android:name="android.software.leanback"
        android:required="false" />
    ...
</manifest>
```

Declare touchscreen not required

Applications that are intended to run on TV devices do not rely on touch screens for input. In order to make this clear, the manifest of your TV app must declare that a the `android.hardware.touchscreen` feature is not required. This setting identifies your app as being able to work on a TV device, and is required for your app to be considered a TV app in Google Play. The following code example shows how to include this manifest declaration:

```
<manifest>
    <uses-feature android:name="android.hardware.touchscreen"
              android:required="false" />
    ...
</manifest>
```

Caution: You must declare that a touch screen is not required in your app manifest, as shown this example code, or your app cannot appear in the Google Play store on TV devices.

Provide a home screen banner

An application must provide a home screen banner for each localization if

it includes a Leanback launcher intent filter. The banner is the app launch point that appears on the home screen in the apps and games rows. Desribe the banner in the manifest as follows:

```
<application
    ...
    android:banner="@drawable/banner" >

    ...
</application>
```

Use the `android:banner` attribute with the `<application>` tag to supply a default banner for all application activities, or with the `<activity>` tag to supply a banner for a specific activity.

See Banners in the UI Patterns for TV design guide.

Add TV Support Libraries

The Android SDK includes support libraries that are intended for use with TV apps. These libraries provide APIs and user interface widgets for use on TV devices. The libraries are located in the `<sdk>/extras/android/support/` directory. Here is a list of the libraries and their general purpose:

- **v17 leanback library - Provides user interface widgets for TV apps, particularly for apps that do media playback.**
- **v7 recyclerview library - Provides classes for managing display of long lists in a**

memory efficient manner. Several classes in the v17 leanback library depend on the classes in this library.

- v7 cardview library - Provides user interface widgets for displaying information cards, such as media item pictures and descriptions.

Note: You are not required to use these support libraries for your TV app. However, we strongly recommend using them, particularly for apps that provide a media catalog browsing interface.

If you decide to use the v17 leanback library for your app, you should note that it is dependent on the v4 support library. This means that apps that use the leanback support library should include all of these support libraries:

- v4 support library
- v7 recyclerview support library
- v17 leanback support library

The v17 leanback library contains resources, which require you to take

specific steps to include it in app projects. For instructions on importing a support library with resources, see Support Library Setup.

Build TV Apps

After you have completed the steps described above, it's time to start building apps for the big screen! Check out these additional topics to help you build your app for TV:

- Building TV Playback Apps - TVs are built to entertain, so Android provides a set of user interface tools and widgets for building TV apps that play videos and music, and let users browse for the content they want.
- Helping Users Find Your Content on TV - With all the content choices at users' fingertips, helping them find content they enjoy is almost as important as providing that content. This training discusses how to surface your content on TV devices.
- Building TV Games - TV devices are a great platform for games. See this topic for information on building great game experiences for TV.
- Building Live TV Apps - Present your video content in a linear, "broadcast TV" style with channels and programs that your users can access through a program guide as well as the channel up/down buttons.

Run TV Apps

Running your app is an important part of the development process. The AVD Manager in the Android SDK provides the device definitions that allow you to create virtual TV devices for running and testing your applications.

To create an virtual TV device:

1. Start the AVD Manager. For more information, see the AVD Manager help.
2. In the AVD Manager dialog, click the **Device Definitions** tab.
3. Select one of the Android TV device definitions and click **Create AVD**.
4. Select the emulator options and click **OK** to create the AVD.

   Note: For best performance of the TV emulator device, enable the **Use Host GPU** option and, where supported, use virtual device acceleration. For more information on hardware acceleration of the emulator, see Using the Emulator.

To test your application on the virtual TV device:

1. Compile your TV application in your development environment.
2. Run the application from your development environment and choose the TV virtual device as the target.

For more information about using emulators see, Using the Emulator. For more information on deploying apps from Android Studio to virtual devices, see Debugging with Android Studio. For more information about deploying apps to emulators from Eclipse with ADT, see Building and Running from Eclipse with ADT.

Next Get started Next class

Handling TV Hardware

Previous Next Get started

This lesson teaches you how to

1. Check for a TV Device
2. Handle Unsupported Hardware Features
3. Manage Hardware Controllers

TV hardware is substantially different from other Android devices. TVs do not include some of the hardware features found on other Android devices, such as touch screens, cameras, and GPS receivers. TVs are also completely dependent on secondary hardware devices. In order for users to interact with TV apps, they must use a remote control or game pad. When you build an app for TV, you must carefully consider the hardware limitations and requirements of operating on TV hardware.

This lesson discusses how to check if your app is running on a TV, how to handle unsupported hardware features, and discusses the requirements for handling controllers for TV devices.

Check for a TV Device

If you are building an app that operates both on TV devices and other devices, you may need to check what kind of device your app is running on and adjust the operation of your app. For instance, if you have an app that can be started through an `Intent`, your application should check the device properties to determine if it should start a TV-oriented activity or a phone activity.

The recommended way to determine if your app is running on a TV device is to use the `UiModeManager.getCurrentModeType()` method to check

if the device is running in television mode. The following example code shows you how to check if your app is running on a TV device:

```
public static final String TAG = "DeviceTypeRuntimeCheck";

UiModeManager uiModeManager = (UiModeManager)
getSystemService(UI_MODE_SERVICE);
if (uiModeManager.getCurrentModeType() ==
Configuration.UI_MODE_TYPE_TELEVISION) {
    Log.d(TAG, "Running on a TV Device")
} else {
    Log.d(TAG, "Running on a non-TV Device")
}
```

Handle Unsupported Hardware Features

Depending on the design and functionality of your app, you may be able to work around certain hardware features being unavailable. This section discusses what hardware features are typically not available for TV, how to detect missing hardware features, and suggests alternatives to using these features.

Unsupported TV hardware features

TVs have a different purpose from other devices, and so they do not have hardware features that other Android-powered devices often have. For this reason, the Android system does not support the following features for a TV device:

Hardware	Android feature descriptor
Touchscreen	`android.hardware.touchscreen`
Touchscreen emulator	`android.hardware.faketouch`
Telephony	`android.hardware.telephony`
Camera	`android.hardware.camera`
Bluetooth	`android.hardware.bluetooth`
Near Field Communications (NFC)	`android.hardware.nfc`
GPS	`android.hardware.location.gps`
Microphone [1]	`android.hardware.microphone`
Sensors	`android.hardware.sensor`

[1] Some TV controllers have a microphone, which is not the same as the microphone hardware feature described here. The controller microphone is fully supported.

See the Features Reference for a complete list of features, subfeatures, and their descriptors.

Declaring hardware requirements for TV

Android apps can declare hardware feature requirements in the app manifest to ensure that they do not get installed on devices that do not provide those features. If you are extending an existing app for use on TV, closely review your app's manifest for any hardware requirement declarations that might prevent it from being installed on a TV device.

If your app uses hardware features (such as a touchscreen or camera) that are not available on TV, but can operate without the use of those features, modify your app's manifest to indicate that these features are not required by your app. The following manifest code snippet demonstrates how to declare that your app does not require hardware features which are unavailable on TV devices, even though your app may use these features on non-TV devices:

```
<uses-feature android:name="android.hardware.touchscreen"
        android:required="false"/>
<uses-feature android:name="android.hardware.faketouch"
        android:required="false"/>
<uses-feature android:name="android.hardware.telephony"
        android:required="false"/>
<uses-feature android:name="android.hardware.camera"
        android:required="false"/>
<uses-feature android:name="android.hardware.bluetooth"
        android:required="false"/>
<uses-feature android:name="android.hardware.nfc"
        android:required="false"/>
<uses-feature android:name="android.hardware.location.gps"
        android:required="false"/>
<uses-feature android:name="android.hardware.microphone"
        android:required="false"/>
<uses-feature android:name="android.hardware.sensor"
        android:required="false"/>
```

Note: Some features have subfeatures like `android.hardware.camera.front`, as described in the Feature Reference.

Be sure to mark as `required="false"` any subfeatures also used in your app.

All apps intended for use on TV devices must declare that the touch screen feature is not required as described in Get Started with TV Apps. If your app normally uses one or more of the features listed above, change the `android:required` attribute setting to `false` for those features in your manifest.

Caution: Declaring a hardware feature as required by setting its value to `true` prevents your app from being installed on TV devices or appearing in the Android TV home screen launcher.

Once you decide to make hardware features optional for your app, you must check for the availability of those features at runtime and then adjust your app's behavior. The next section discusses how to check for hardware features and suggests some approaches for changing the behavior of your app.

For more information on filtering and declaring features in the manifest, see the `uses-feature` guide.

Declaring permissions that imply hardware features

Some `uses-permission` manifest declarations *imply hardware features*. This behavior means that requesting some permissions in your app manifest can exclude your app from from being installed and used on TV devices. The following commonly requested permissions create an implicit hardware feature requirement:

Permission	Implied hardware feature
RECORD_AUDIO	`android.hardware.microphone`
CAMERA	`android.hardware.camera` *and* `android.hardware.camera.autofocus`
ACCESS_COARSE_LOCATION	`android.hardware.location` *and* `android.hardware.location.network`
ACCESS_FINE_LOCATION	`android.hardware.location` *and*

`android.hardware.location.gps`

For a complete list of permission requests that imply a hardware feature requirement, see the `uses-feature` guide. If your app requests one of the features listed above, include a `uses-feature` declaration in your manifest for the implied hardware feature that indicates it is not required (`android:required="false"`).

Checking for hardware features

The Android framework can tell you if hardware features are not available on the device where your app is running. Use the `hasSystemFeature(String)` method to check for specific features at runtime. This method takes a single string argument that specifies the feature you want to check.

The following code example demonstrates how to detect the availability of hardware features at runtime:

```
// Check if the telephony hardware feature is available.
if
(getPackageManager().hasSystemFeature("android.hardware.tel
{
    Log.d("HardwareFeatureTest", "Device can make phone
calls");
}

// Check if android.hardware.touchscreen feature is
available.
if
(getPackageManager().hasSystemFeature("android.hardware.tou
{
    Log.d("HardwareFeatureTest", "Device has a touch
screen.");
}
```

Touch screen

Since most TVs do not have touch screens, Android does not support

touch screen interaction for TV devices. Furthermore, using a touch screen is not consistent with a viewing environment where the user is seated 10 feet away from the display. Make sure that your UI elements and text do not require or imply the use of a touchscreen.

On TV devices, you should design your app to work with this interaction model by supporting navigation using a directional pad (D-pad) on a TV remote control. For more information on properly supporting navigation using TV-friendly controls, see Creating TV Navigation.

Camera

Although a TV typically does not have a camera, you can still provide a photography-related app on a TV. For example, if you have an app that takes, views, and edits photos, you can disable its picture-taking functionality for TVs and still allow users to view and even edit photos. If you decide to enable your camera-related app to work on a TV, add the following feature declaration your app manifest:

```
<uses-feature android:name="android.hardware.camera"
android:required="false" />
```

If you enable your app to run without a camera, add code to your app that detects if the camera feature is available and makes adjustments to the operation of your app. The following code example demonstrates how to detect the presence of a camera:

```
// Check if the camera hardware feature is available.
if (getPackageManager().hasSystemFeature("android.hardware.camera"))
{
    Log.d("Camera test", "Camera available!");
} else {
    Log.d("Camera test", "No camera available. View and edit
features only.");
}
```

GPS

TVs are stationary, indoor devices, and do not have built-in global positioning system (GPS) receivers. If your app uses location information, you can still allow users to search for a location, or use a static location

provider such as a zip code configured during the TV device setup.

```
// Request a static location from the location manager
LocationManager locationManager = (LocationManager)
this.getSystemService(
        Context.LOCATION_SERVICE);
Location location = locationManager.getLastKnownLocation("static");

// Attempt to get postal or zip code from the static location object
Geocoder geocoder = new Geocoder(this);
Address address = null;
try {
  address = geocoder.getFromLocation(location.getLatitude(),
          location.getLongitude(), 1).get(0);
  Log.d("Zip code", address.getPostalCode());

} catch (IOException e) {
  Log.e(TAG, "Geocoder error", e);
}
```

Handling Controllers

TV devices require a secondary hardware device for interacting with apps, in the form of a basic remote controller or game controller. This means that your app must support D-pad input. It also means that your app may need to handle controllers going offline and input from more than one type of controller.

D-pad minimum controls

The default controller for a TV device is a D-pad. In general, your app should be operable from a remote controller that only has up, down, left, right, select, Back, and Home buttons. If your app is a game that typically requires a game controller with additional controls, your app should attempt to allow gameplay with these D-pad controls. In this case, your app should also warn the user that a controller is required and allow them to exit your game gracefully using the D-pad controller. For more information about handling navigation with D-pad controller for TV devices, see Creating TV Navigation.

Handle controller disconnects

Controllers for TV are frequently Bluetooth devices which may attempt to

save power by periodically going into sleep mode and disconnecting from the TV device. This means that an app might be interrupted or restarted if it is not configured to handle these reconnect events. These events can happen in any of the following circumstances:

- While watching a video which is several minutes long, a D-Pad or game controller goes into sleep mode, disconnects from the TV device and then reconnects later on.
- During gameplay, a new player joins the game using a game controller that is not currently connected.
- During gameplay, a player leaves the game and disconnects a game controller.

Any TV app activity that is subject to disconnect and reconnect events must be configured to handle reconnection events in the app manifest. The following code sample demonstrates how to enable an activity to handle configuration changes, including a keyboard or navigation device connecting, disconnecting, or reconnecting:

```
<activity
  android:name="com.example.android.TvActivity"
  android:label="@string/app_name"
  android:configChanges="keyboard|keyboardHidden|navigation"
  android:theme="@style/Theme.Leanback">

  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category
android:name="android.intent.category.LEANBACK_LAUNCHER" />
  </intent-filter>
  ...
</activity>
```

This configuration change allows the app to continue running through a reconnection event, rather than being restarted by the Android framework, which is not a good user experience.

Handle D-pad input variations

TV device users may have more than one type of controller that they use with their TV. For example, a user might have both a basic D-pad controller and a game controller. The key codes provided by a game controller when it is being used for D-pad functions may vary from the

key codes sent by a physical D-pad.

Your app should handle the variations of D-pad input from a game controller, so the user does not have to physically switch controllers to operate your app. For more information on handling these input variations, see Handling Controller Actions.

Next Get started Next class

Building Layouts for TV

Previous Next Get started

This lesson teaches you how to

1. Use Layout Themes for TV
2. Build Basic TV Layouts
3. Build Useable Text and Controls
4. Manage Layout Resources for TV
5. Avoid Layout Anti-Patterns
6. Handle Large Bitmaps
7. Provide Effective Advertising

You should also read

1. Android TV Design

A TV screen is typically viewed from about 10 feet away, and while it is much larger than most other Android device displays, this type of screen does not provide the same level of precise detail and color as a smaller device. These factors require you to create app layouts with TV devices in mind in order to create a useful and enjoyable user experience.

This lesson describes the minimum requirements and implementation details for building effective layouts in TV apps.

Use Layout Themes for TV

Android Themes can provide a basis for layouts in your TV apps. You should use a theme to modify the display of your app activities that are meant to run on a TV device. This section explains which themes you should use.

Leanback theme

A support library for TV user interfaces called the v17 leanback library provides a standard theme for TV activities, called `Theme.Leanback`. This theme establishes a consistent visual style for TV apps. Use of this theme is recommended for most TV apps. This theme is strongly recommended for any TV app that uses v17 leanback classes. The following code sample shows how to apply this theme to a given activity within an app:

```
<activity
  android:name="com.example.android.TvActivity"
  android:label="@string/app_name"
  android:theme="@style/Theme.Leanback">
```

NoTitleBar theme

The title bar is a standard user interface element for Android apps on phones and tablets, but it is not appropriate for TV apps. If you are not using v17 leanback classes, you should apply this theme to your TV activities to suppress the display of a title bar. The following code example from a TV app manifest demonstrates how to apply this theme to remove the display of a title bar:

```
<application>
  ...

  <activity
    android:name="com.example.android.TvActivity"
    android:label="@string/app_name"
    android:theme="@android:style/Theme.NoTitleBar">
    ...

  </activity>
</application>
```

Build Basic TV Layouts

Layouts for TV devices should follow some basic guidelines to ensure they are usable and effective on large screens. Follow these tips to build landscape layouts optimized for TV screens:

- Build layouts with a landscape orientation. TV screens always display

in landscape mode.

- Put on-screen navigation controls on the left or right side of the screen and save the vertical space for content.
- Create UIs that are divided into sections, using Fragments, and use view groups like `GridView` instead of `ListView` to make better use of the horizontal screen space.
- Use view groups such as `RelativeLayout` or `LinearLayout` to arrange views. This approach allows the system to adjust the position of the views to the size, alignment, aspect ratio, and pixel density of a TV screen.
- Add sufficient margins between layout controls to avoid a cluttered UI.

Overscan

Layouts for TV have some unique requirements due to the evolution of TV standards and the desire to always present a full screen picture to viewers. For this reason, TV devices may clip the outside edge of an app layout in order to ensure that the entire display is filled. This behavior is generally referred to as *overscan*.

Avoid screen elements being clipped due to overscan and by incorporating a 10% margin on all sides of your layout. This translates into a 48dp margin on the left and right edges and a 27dp margin on the top and bottom of your base layouts for activities. The following example layout demonstrates how to set these margins in the root layout for a TV app:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/base_layout"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical"
  android:layout_marginTop="27dp"
  android:layout_marginLeft="48dp"
  android:layout_marginRight="48dp"
  android:layout_marginBottom="27dp" >
</LinearLayout>
```

Caution: Do not apply overscan margins to your layout if you are using

the v17 leanback classes, such as `BrowseFragment` or related widgets, as those layouts already incorporate overscan-safe margins.

Build Useable Text and Controls

The text and controls in a TV app layout should be easily visible and navigable from a distance. Follow these tips to make your user interface elements easier to see from a distance:

- Break text into small chunks that users can quickly scan.
- Use light text on a dark background. This style is easier to read on a TV.
- Avoid lightweight fonts or fonts that have both very narrow and very broad strokes. Use simple sans-serif fonts and anti-aliasing to increase readability.
- Use Android's standard font sizes:

```
<TextView
      android:id="@+id/atext"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:gravity="center_vertical"
      android:singleLine="true"
      android:textAppearance="?
android:attr/textAppearanceMedium"/>
```

- Ensure that all your view widgets are large enough to be clearly visible to someone sitting 10 feet away from the screen (this distance is greater for very large screens). The best way to do this is to use layout-relative sizing rather than absolute sizing, and density-independent pixel (dip) units instead of absolute pixel units. For example, to set the width of a widget, use `wrap_content` instead of a pixel measurement, and to set the margin for a widget, use dip values instead of px values.

For more information about density-independent pixels and building layouts to handle larger screen sizes, see Supporting Multiple Screens.

Manage Layout Resources for TV

The common high-definition TV display resolutions are 720p, 1080i, and 1080p. Your TV layout should target a screen size of 1920 x 1080 pixels, and then allow the Android system to downscale your layout elements to 720p if necessary. In general, downscaling (removing pixels) does not degrade your layout presentation quality. However, upscaling can cause display artifacts that degrade the quality of your layout and have a negative impact on the user experience of your app.

To get the best scaling results for images, provide them as 9-patch image elements if possible. If you provide low quality or small images in your layouts, they will appear pixelated, fuzzy, or grainy, which is not a good experience for the user. Use high-quality images instead.

For more information on optimizing layouts and resources for large screens see Designing for multiple screens.

Avoid Layout Anti-Patterns

There are a few approaches to building layouts that you should avoid because they do not work well on TV devices and lead to bad user experiences. Here are some user interface approaches you should specifically *not* use when developing a layout for TV.

- **Re-using phone or tablet layouts** - Do not reuse layouts from a phone or tablet app without modification. Layouts built for other Android device form factors are not well suited for TV devices and should be simplified for operation on a TV.
- **ActionBar** - While this user interface convention is recommended for use on phones and tablets, it is not appropriate for a TV interface. In particular, using an action bar options menu (or any pull-down menu for that matter) is strongly discouraged, due to the difficulty in navigating such a menu with a remote control.
- **ViewPager** - Sliding between screens can work great on a phone or tablet, but don't try this on a TV!

For more information on designing layouts that are appropriate to TV, see the TV Design guide.

Handle Large Bitmaps

TV devices, like any other Android device, have a limited amount of memory. If you build your app layout with very high-resolution images or use many high-resolution images in the operation of your app, it can quickly run into memory limits and cause out of memory errors. To avoid these types of problems, follow these tips:

- Load images only when they are displayed on the screen. For example, when displaying multiple images in a `GridView` or `Gallery`, only load an image when `getView()` is called on the view's `Adapter`.
- Call `recycle()` on `Bitmap` views that are no longer needed.
- Use `WeakReference` for storing references to `Bitmap` objects in an in-memory `Collection`.
- If you fetch images from the network, use `AsyncTask` to fetch and store them on the device for faster access. Never do network transactions on the application's main user interface thread.
- Scale down large images to a more appropriate size as you download them; otherwise, downloading the image itself may cause an out of memory exception.

For more information on getting the best performance when working with images, see Displaying Bitmaps Efficiently.

Provide Effective Advertising

For the living room environment, we recommend you use video ads solutions that are full-screen and dismissable within 30 seconds. Functionality for advertising on Android TV, such as dismiss buttons and clickthroughs, must be accessible using the D-pad rather than touch.

Android TV does not provide a web browser. Your ads must not attempt to launch a web browser or redirect to Google Play Store content that is not approved for Android TV devices.

Note: You can use the `WebView` class for logins to services like Google+ and Facebook.

Next Get started Next class

Creating TV Navigation

Previous Next Get started

This lesson teaches you how to

1. Enable D-pad Navigation
2. Provide Clear Focus and Selection

TV devices provide a limited set of navigation controls for apps. Creating an effective navigation scheme for your TV app depends on understanding these limited controls and the limits of users' perception while operating your app. As you build your Android app for TVs, pay special attention to how the user actually navigates around your app when using remote control buttons instead of a touch screen.

This lesson explains the minimum requirements for creating effective TV app navigation scheme and how to apply those requirements to your app.

Enable D-pad Navigation

On a TV device, users navigate with controls on a remote control device, using either a directional pad (D-pad) or arrow keys. This type of control limits movement to up, down, left, and right. To build a great TV-optimized app, you must provide a navigation scheme where the user can quickly learn how to navigate your app using these limited controls.

The Android framework handles directional navigation between layout elements automatically, so you typically do not need to do anything extra for your app. However, you should thoroughly test navigation with a D-pad controller to discover any navigation problems. Follow these guidelines to test that your app's navigation system works well with a D-

pad on a TV device:

- Ensure that a user with a D-pad controller can navigate to all visible controls on the screen.
- For scrolling lists with focus, make sure that the D-pad up and down keys scroll the list, and the Enter key selects an item in the list. Verify that users can select an element in the list and that the list still scrolls when an element is selected.
- Ensure that switching between controls between controls is straightforward and predictable.

Modifying directional navigation

The Android framework automatically applies a directional navigation scheme based on the relative position of focusable elements in your layouts. You should test the generated navigation scheme in your app using a D-pad controller. After testing, if you decide you want users to move through your layouts in a specific way, you can set up explicit directional navigation for your controls.

Note: You should only use these attributes to modify the navigation order if the default order that the system applies does not work well.

The following code sample shows how to define the next control to receive focus for a `TextView` layout object:

```
<TextView android:id="@+id/Category1"
        android:nextFocusDown="@+id/Category2"\>
```

The following table lists all of the available navigation attributes for Android user interface widgets:

Attribute	Function
`nextFocusDown`	Defines the next view to receive focus when the user navigates down.
`nextFocusLeft`	Defines the next view to receive focus when the user navigates left.
`nextFocusRight`	Defines the next view to receive focus when the user

	navigates right.
`nextFocusUp`	Defines the next view to receive focus when the user navigates up.

To use one of these explicit navigation attributes, set the value to the ID (`android:id` value) of another widget in the layout. You should set up the navigation order as a loop, so that the last control directs focus back to the first one.

Provide Clear Focus and Selection

The success of an app's navigation scheme on TV devices is depends on how easy it is for a user to determine what user interface element is in focus on screen. If you do not provide clear indications of focused items (and therefore what item a user can take action on), they can quickly become frustrated and exit your app. For the same reason, it is important to always have an item in focus that a user can take action on immediately after your app starts, or any time it is idle.

Your app layout and implementation should use color, size, animation, or a combination of these attributes to help users easily determine what actions they can take next. Use a uniform scheme for indicating focus across your application.

Android provides Drawable State List Resources to implement highlights for focused and selected controls. The following code example demonstrates how to enable visual behavior for a button to indicate that a user has navigated to the control and then selected it:

```
<!-- res/drawable/button.xml -->
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:state_pressed="true"
          android:drawable="@drawable/button_pressed" /> <!--
pressed -->
    <item android:state_focused="true"
          android:drawable="@drawable/button_focused" /> <!--
focused -->
    <item android:state_hovered="true"
          android:drawable="@drawable/button_focused" /> <!--
hovered -->
    <item android:drawable="@drawable/button_normal" /> <!-- default
```

```
-->
</selector>
```

The following layout XML sample code applies the previous state list drawable to a `Button`:

```
<Button
    android:layout_height="wrap_content"
    android:layout_width="wrap_content"
    android:background="@drawable/button" />
```

Make sure to provide sufficient padding within the focusable and selectable controls so that the highlights around them are clearly visible.

For more recommendations on designing effective selection and focus for your TV app, see Patterns for TV.

Next Get started Next class

Building TV Playback Apps

Previous Next Get started

Dependencies and Prerequisites

- Android 5.0 (API level 21) or higher

You should also read

- Design for TV

Video

DevBytes: Android TV — Using the Leanback library

Browsing and playing media files is frequently part of the user experience provided by a TV app. Building such an experience from scratch, while making sure that it is fast, fluid, and attractive can be quite challenging. Whether your app provides access to a small or large media catalog, it is important to allow users to quickly browse options and get to the content they want.

The Android framework provides classes for building user interfaces for these types of apps with the v17 leanback support library. This library provides a framework of classes for creating an efficient and familiar interface for browsing and playing media files with minimal coding. The classes are designed to be extended and customized so you can create an experience that is unique to your app.

This class shows you how to build a TV app for browsing and playing media content using the Leanback support libraries for TV.

Topics

Creating a Catalog Browser
Learn how to use the Leanback support library to build a browsing interface for media catalogs.

Providing a Card View
Learn how to use the Leanback support library to build a card view for content items.

Building a Details View
Learn how to use the Leanback support library to build a details page for media items.

Displaying a Now Playing Card
Learn how to use a MediaSession to display a Now Playing card on the home screen.

Next Get started Next class

Creating a Catalog Browser

Previous Next Get started

This lesson teaches you to

1. Create a Media Browse Layout
2. Display Media Lists
3. Update the Background

Media apps that run on TV need to allow users to browse its content offerings, make a selection, and start playing content. The content browsing experience for apps of this type should be simple and intuitive, as well as visually pleasing and engaging.

This lesson discusses how to use the classes provided by the v17 leanback support library to implement a user interface for browsing music or videos from your app's media catalog.

Create a Media Browse Layout

The `BrowseFragment` class in the leanback library allows you to create a primary layout for browsing categories and rows of media items with a minimum of code. The following example shows how to create a layout that contains a `BrowseFragment`:

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical"
  >

  <fragment
      android:name="android.support.v17.leanback.app.BrowseFragment"
      android:id="@+id/browse_fragment"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
```

```
        />
</LinearLayout>
```

In order to work with this layout in an activity, retrieve the `BrowseFragment` element from the layout. Use the methods in this class to set display parameters such as the icon, title, and whether category headers are enabled. The following code sample demonstrates how to set the layout parameters for a `BrowseFragment` in a layout:

```
public class BrowseMediaActivity extends Activity {

    public static final String TAG ="BrowseActivity";

    protected BrowseFragment mBrowseFragment;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.browse_fragment);

        final FragmentManager fragmentManager = getFragmentManager();
        mBrowseFragment = (BrowseFragment)
fragmentManager.findFragmentById(
                R.id.browse_fragment);

        // Set display parameters for the BrowseFragment

mBrowseFragment.setHeadersState(BrowseFragment.HEADERS_ENABLED);
        mBrowseFragment.setTitle(getString(R.string.app_name));
        mBrowseFragment.setBadgeDrawable(getResources().getDrawable(
                R.drawable.ic_launcher));
        mBrowseFragment.setBrowseParams(params);

    }
}
```

Displaying Media Lists

The `BrowseFragment` allows you to define and display browsable media content categories and media items from a media catalog using adapters and presenters. Adapters enable you to connect to local or online data sources that contain your media catalog information. Presenters hold data about media items and provide layout information for displaying an item on screen.

The following example code shows an implementation of a `Presenter` for displaying string data:

```
public class StringPresenter extends Presenter {
```

```
    private static final String TAG = "StringPresenter";

    public ViewHolder onCreateViewHolder(ViewGroup parent) {
        TextView textView = new TextView(parent.getContext());
        textView.setFocusable(true);
        textView.setFocusableInTouchMode(true);
        textView.setBackground(

parent.getContext().getResources().getDrawable(R.drawable.text_bg));
        return new ViewHolder(textView);
    }

    public void onBindViewHolder(ViewHolder viewHolder, Object item)
{
        ((TextView) viewHolder.view).setText(item.toString());
    }

    public void onUnbindViewHolder(ViewHolder viewHolder) {
        // no op
    }
}
```

Once you have constructed a presenter class for your media items, you can build and attach an adapter to the `BrowseFragment` to display those items on screen for browsing by the user. The following example code demonstrates how to construct an adapter to display categories and items in those categories using the `StringPresenter` class shown in the previous code example:

```
private ArrayObjectAdapter mRowsAdapter;
private static final int NUM_ROWS = 4;

@Override
protected void onCreate(Bundle savedInstanceState) {
    ...

    buildRowsAdapter();
}

private void buildRowsAdapter() {
    mRowsAdapter = new ArrayObjectAdapter(new ListRowPresenter());

    for (int i = 0; i < NUM_ROWS; ++i) {
        ArrayObjectAdapter listRowAdapter = new ArrayObjectAdapter(
                new StringPresenter());
        listRowAdapter.add("Media Item 1");
        listRowAdapter.add("Media Item 2");
        listRowAdapter.add("Media Item 3");
        HeaderItem header = new HeaderItem(i, "Category " + i,
null);
        mRowsAdapter.add(new ListRow(header, listRowAdapter));
    }

    mBrowseFragment.setAdapter(mRowsAdapter);
}
```

This example shows a static implementation of the adapters. A typical media browsing application uses data from an online database or web service. For an example of a browsing application that uses data retrieved from the web, see the Android TV sample app.

Update the Background

In order to add visual interest to a media-browsing app on TV, you can update the background image as users browse through content. This technique can make interaction with your app feel more cinematic and enjoyable for users.

The Leanback support library provides a `BackgroundManager` class for changing the background of your TV app activity. The following example shows how to create a simple method for updating the background within your TV app activity:

```
protected void updateBackground(Drawable drawable) {
    BackgroundManager.getInstance(this).setDrawable(drawable);
}
```

Many of the existing media-browse apps automatically update the background as the user navigates through media listings. In order to do this, you can set up a selection listener to automatically update the background based on the user's current selection. The following example shows you how to set up an `OnItemViewSelectedListener` class to catch selection events and update the background:

```
protected void clearBackground() {

BackgroundManager.getInstance(this).setDrawable(mDefaultBackground);
}

protected OnItemViewSelectedListener
getDefaultItemViewSelectedListener() {
    return new OnItemViewSelectedListener() {
        @Override
        public void onItemSelected(Object item, Row row) {
            if (item instanceof Movie ) {
                URI uri = ((Movie)item).getBackdropURI();
                updateBackground(uri);
            } else {
                clearBackground();
            }
        }
```

```
    };
}
```

Note: The implementation above is a simple example shown for purposes of illustration. When creating this function in your own app, you should consider running the background update action in a separate thread for better performance. In addition, if you are planning on updating the background in response to users scrolling through items, consider adding a time to delay a background image update until the user settles on an item. This technique avoids excessive background image updates.

Next Get started Next class

Providing a Card View

Previous Next Get started

This lesson teaches you to

1. Create a Card Presenter
2. Create a Card View

Try it out

- Android Leanback sample app

In the previous lesson, you created a catalog browser, implemented in a browse fragment, that displays a list of media items. In this lesson, you create the card views for your media items and present them in the browse fragment.

The `BaseCardView` class and subclasses display the meta data associated with a media item. The `ImageCardView` class used in this lesson displays an image for the content along with the media item's title.

This lesson describes code from the Android Leanback sample app, available on GitHub. Use this sample code to start your own app.

Figure 1. The Leanback sample app browse fragment with a card presenter displaying card view objects.

Create a Card Presenter

A `Presenter` generates views and binds objects to them on demand. In the browse fragment where your app presents its content to the user, you create a `Presenter` for the content cards and pass it to the adapter that adds the content to the screen. In the following code, the `CardPresenter` is created in the `onLoadFinished()` callback of the `LoaderManager`.

```
@Override
public void onLoadFinished(Loader<HashMap<String, List<Movie>>> arg0,
                           HashMap<String, List<Movie>> data) {

    mRowsAdapter = new ArrayObjectAdapter(new ListRowPresenter());
    CardPresenter cardPresenter = new CardPresenter();

    int i = 0;

    for (Map.Entry<String, List<Movie>> entry : data.entrySet()) {
        ArrayObjectAdapter listRowAdapter = new
ArrayObjectAdapter(cardPresenter);
        List<Movie> list = entry.getValue();

        for (int j = 0; j < list.size(); j++) {
            listRowAdapter.add(list.get(j));
```

```
        }
        HeaderItem header = new HeaderItem(i, entry.getKey(), null);
        i++;
        mRowsAdapter.add(new ListRow(header, listRowAdapter));
    }

    HeaderItem gridHeader = new HeaderItem(i,
getString(R.string.more_samples),
            null);

    GridItemPresenter gridPresenter = new GridItemPresenter();
    ArrayObjectAdapter gridRowAdapter = new
ArrayObjectAdapter(gridPresenter);
    gridRowAdapter.add(getString(R.string.grid_view));
    gridRowAdapter.add(getString(R.string.error_fragment));
    gridRowAdapter.add(getString(R.string.personal_settings));
    mRowsAdapter.add(new ListRow(gridHeader, gridRowAdapter));

    setAdapter(mRowsAdapter);

    updateRecommendations();
}
```

Create a Card View

In this step, you build the card presenter with a view holder for the card view that describes your media content items. Note that each presenter must only create one view type. If you have two different card view types then you need two different card presenters.

In the `Presenter`, implement an `onCreateViewHolder()` callback that creates a view holder that can be used to display a content item.

```
@Override
public class CardPresenter extends Presenter {

    private Context mContext;
    private static int CARD_WIDTH = 313;
    private static int CARD_HEIGHT = 176;
    private Drawable mDefaultCardImage;

    @Override
    public ViewHolder onCreateViewHolder(ViewGroup parent) {
        mContext = parent.getContext();
        mDefaultCardImage =
mContext.getResources().getDrawable(R.drawable.movie);
...
```

In the `onCreateViewHolder()` method, create a card view for content items. The sample below uses an `ImageCardView`.

When a card is selected, the default behavior expands it to a larger size.

If you want to designate a different color for the selected card, call `setSelected()` as shown here.

```
...
    ImageCardView cardView = new ImageCardView(mContext) {
        @Override
        public void setSelected(boolean selected) {
            int selected_background =
mContext.getResources().getColor(R.color.detail_background);
            int default_background =
mContext.getResources().getColor(R.color.default_background);
            int color = selected ? selected_background :
default_background;
            findViewById(R.id.info_field).setBackgroundColor(color);
            super.setSelected(selected);
        }
    };
...
```

When the user opens your app, the `Presenter.ViewHolder` displays the `CardView` objects for your content items. You need to set these to receive focus from the D-pad controller by calling `setFocusable(true)` and `setFocusableInTouchMode(true)`.

```
...
    cardView.setFocusable(true);
    cardView.setFocusableInTouchMode(true);
    return new ViewHolder(cardView);
}
```

When the user selects the `ImageCardView`, it expands to reveal its text area with the background color you specify, as shown in figure 2.

Figure 2. The Leanback sample app image card view when selected.

Next Get started Next class

Building a Details View

Previous Next Get started

This lesson teaches you to

1. Build a Details Presenter
2. Extend the Details Fragment
3. Create a Details Activity
4. Define a Listener for Clicked Items

The media browsing interface classes provided by the v17 leanback support library include classes for displaying additional information about a media item, such as a description or reviews, and for taking action on that item, such as purchasing it or playing its content.

This lesson discusses how to create a presenter class for media item details, and how to extend the `DetailsFragment` class to implement a details view for a media item when it is selected by a user.

Note: The implementation example shown here uses an additional activity to contain the `DetailsFragment`. However, it is possible to avoid creating a second activity by replacing the current `BrowseFragment` with a `DetailsFragment` within the *same* activity using fragment transactions. For more information on using fragment transactions, see the Building a Dynamic UI with Fragments training.

Build a Details Presenter

In the media browsing framework provided by the leanback library, you use presenter objects to control the display of data on screen, including media item details. The framework provides the `AbstractDetailsDescriptionPresenter` class for this purpose, which is a

nearly complete implementation of the presenter for media item details. All you have to do is implement the `onBindDescription()` method to bind the view fields to your data objects, as shown in the following code sample:

```
public class DetailsDescriptionPresenter
        extends AbstractDetailsDescriptionPresenter {

    @Override
    protected void onBindDescription(ViewHolder viewHolder, Object
itemData) {
        MyMediaItemDetails details = (MyMediaItemDetails) itemData;
        // In a production app, the itemData object contains the
information
        // needed to display details for the media item:
        // viewHolder.getTitle().setText(details.getShortTitle());

        // Here we provide static data for testing purposes:
        viewHolder.getTitle().setText(itemData.toString());
        viewHolder.getSubtitle().setText("2014   Drama   TV-14");
        viewHolder.getBody().setText("Lorem ipsum dolor sit amet,
consectetur "
                + "adipisicing elit, sed do eiusmod tempor
incididunt ut labore "
                + " et dolore magna aliqua. Ut enim ad minim veniam,
quis "
                + "nostrud exercitation ullamco laboris nisi ut
aliquip ex ea "
                + "commodo consequat.");
    }
}
```

Extend the Details Fragment

When using the `DetailsFragment` class for displaying your media item details, extend that class to provide additional content such as a preview image and actions for the media item. You can also provide additional content, such as a list of related media items.

The following example code demonstrates how to use the presenter class shown in the previous section, to add a preview image and actions for the media item being viewed. This example also shows the addition of a related media items row, which appears below the details listing.

```
public class MediaItemDetailsFragment extends DetailsFragment {
    private static final String TAG = "MediaItemDetailsFragment";
    private ArrayObjectAdapter mRowsAdapter;

    @Override
    public void onCreate(Bundle savedInstanceState) {
```

```
        Log.i(TAG, "onCreate");
        super.onCreate(savedInstanceState);

        buildDetails();
    }

    private void buildDetails() {
        ClassPresenterSelector selector = new
ClassPresenterSelector();
        // Attach your media item details presenter to the row
presenter:
        DetailsOverviewRowPresenter rowPresenter =
            new DetailsOverviewRowPresenter(new
DetailsDescriptionPresenter());

        selector.addClassPresenter(DetailsOverviewRow.class,
rowPresenter);
        selector.addClassPresenter(ListRow.class,
                new ListRowPresenter());
        mRowsAdapter = new ArrayObjectAdapter(selector);

        Resources res = getActivity().getResources();
        DetailsOverviewRow detailsOverview = new DetailsOverviewRow(
                "Media Item Details");

        // Add images and action buttons to the details view

detailsOverview.setImageDrawable(res.getDrawable(R.drawable.jelly_beans

        detailsOverview.addAction(new Action(1, "Buy $9.99"));
        detailsOverview.addAction(new Action(2, "Rent $2.99"));
        mRowsAdapter.add(detailsOverview);

        // Add a Related items row
        ArrayObjectAdapter listRowAdapter = new ArrayObjectAdapter(
                new StringPresenter());
        listRowAdapter.add("Media Item 1");
        listRowAdapter.add("Media Item 2");
        listRowAdapter.add("Media Item 3");
        HeaderItem header = new HeaderItem(0, "Related Items",
null);
        mRowsAdapter.add(new ListRow(header, listRowAdapter));

        setAdapter(mRowsAdapter);
    }
}
```

Create a Details Activity

Fragments such as the `DetailsFragment` must be contained within an activity in order to be used for display. Creating an activity for your details view, separate from the browse activity, enables you to invoke your details view using an `Intent`. This section explains how to build an activity to contain your implementation of the detail view for your media items.

Start creating the details activity by building a layout that references your implementation of the `DetailsFragment`:

```
<!-- file: res/layout/details.xml -->

<fragment xmlns:android="http://schemas.android.com/apk/res/android"

android:name="com.example.android.mediabrowser.MediaItemDetailsFragment

    android:id="@+id/details_fragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
/>
```

Next, create an activity class that uses the layout shown in the previous code example:

```
public class DetailsActivity extends Activity
{
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.details);
    }
}
```

Finally, add this new activity to the manifest. Remember to apply the Leanback theme to ensure that the user interface is consistent with the media browse activity:

```
<application>
  ...

  <activity android:name=".DetailsActivity"
    android:exported="true"
    android:theme="@style/Theme.Leanback"/>

</application>
```

Define a Listener for Clicked Items

After you have implemented the `DetailsFragment`, modify your main media browsing view to move to your details view when a user clicks on a media item. In order to enable this behavior, add an `OnItemViewClickedListener` object to the `BrowseFragment` that fires an intent to start the item details activity.

The following example shows how to implement a listener to start the

details view when a user clicks a media item in the main media browsing activity:

```
public class BrowseMediaActivity extends Activity {
    ...

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...

        // create the media item rows
        buildRowsAdapter();

        // add a listener for selected items
        mBrowseFragment.OnItemViewClickedListener(
            new OnItemViewClickedListener() {
                @Override
                public void onItemClicked(Object item, Row row) {
                    System.out.println("Media Item clicked: " +
item.toString());
                    Intent intent = new
Intent(BrowseMediaActivity.this,
                            DetailsActivity.class);
                    // pass the item information
                    intent.getExtras().putLong("id", item.getId());
                    startActivity(intent);
                }
            });
    }
}
```

Next Get started Next class

Displaying a Now Playing Card

Previous Next Get started

This lesson teaches you to

1. Start a Media Session
2. Display a Now Playing Card
3. Update the Playback State
4. Respond to User Action

TV apps may allow users to play music or other media in the background while using other applications. If your app allows this type of use, it must must provide a means for the user to return to the app to pause the music or switch to a new song. The Android framework enables TV apps to do this by displaying a *Now Playing* card on the home screen in the recommendations row.

The Now Playing card is a system artifact that displays on the home screen in the recommendations row for an active media session. It includes the media metadata such as the album art, title, and app icon. When the user selects it, the system opens the the app that owns the session.

This lesson shows how to use the `MediaSession` class to implement the Now Playing card.

Start a Media Session

A playback app can run as an activity or as a service. The service is required for background playback because it can continue to play media even after the activity that launched it has been destroyed. For this discussion, the media playback app is assumed to be running in a

`MediaBrowserService`.

In your service's `onCreate()` method, create a new `MediaSession`, set the callback and flags appropriate to a media app, and set the session token for the `MediaBrowserService`.

```
mSession = new MediaSession(this, "MusicService");
mSession.setCallback(new MediaSessionCallback());
mSession.setFlags(MediaSession.FLAG_HANDLES_MEDIA_BUTTONS |
        MediaSession.FLAG_HANDLES_TRANSPORT_CONTROLS);

// for the MediaBrowserService
setSessionToken(mSession.getSessionToken());
```

Note: The Now Playing card will display only for a media session with the `FLAG_HANDLES_TRANSPORT_CONTROLS` flag set.

Display a Now Playing Card

The Now Playing card shows up after `setActive(true)` is called, if the session is the highest priority session in the system. Also, note that your app must request the audio focus, as described in Managing Audio Focus.

```
private void handlePlayRequest() {

    tryToGetAudioFocus();

    if (!mSession.isActive()) {
        mSession.setActive(true);
    }
...
```

The card is removed from the home screen when `setActive(false)` is called or if another app initiates media playback. You may want to remove the card from the home screen some time after playback is paused, depending on how long you want to keep the card up, usually 5 to 30 minutes.

Update the Playback State

As with any media app, update the playback state in the `MediaSession` so that the card can display the current metadata, as shown in the following example:

```
private void updatePlaybackState() {
    long position = PlaybackState.PLAYBACK_POSITION_UNKNOWN;
    if (mMediaPlayer != null && mMediaPlayer.isPlaying()) {
        position = mMediaPlayer.getCurrentPosition();
    }
    PlaybackState.Builder stateBuilder = new PlaybackState.Builder()
            .setActions(getAvailableActions());
    stateBuilder.setState(mState, position, 1.0f);
    mSession.setPlaybackState(stateBuilder.build());
}
private long getAvailableActions() {
    long actions = PlaybackState.ACTION_PLAY |
            PlaybackState.ACTION_PLAY_FROM_MEDIA_ID |
            PlaybackState.ACTION_PLAY_FROM_SEARCH;
    if (mPlayingQueue == null || mPlayingQueue.isEmpty()) {
        return actions;
    }
    if (mState == PlaybackState.STATE_PLAYING) {
        actions |= PlaybackState.ACTION_PAUSE;
    }
    if (mCurrentIndexOnQueue > 0) {
        actions |= PlaybackState.ACTION_SKIP_TO_PREVIOUS;
    }
    if (mCurrentIndexOnQueue < mPlayingQueue.size() - 1) {
        actions |= PlaybackState.ACTION_SKIP_TO_NEXT;
    }
    return actions;
}
```

Display the Media Metadata

For the track currently playing, set the `MediaMetadata` with the `setMetadata()` method. This method of the media session object lets you provide information to the Now Playing card about the track such as the title, subtitle, and various icons. The following example assumes your track's data is stored in a custom data class, `MediaData`.

```
private void updateMetadata(MediaData myData) {
    MediaMetadata.Builder metadataBuilder = new
MediaMetadata.Builder();
    // To provide most control over how an item is displayed set the
    // display fields in the metadata

metadataBuilder.putString(MediaMetadata.METADATA_KEY_DISPLAY_TITLE,
            myData.displayTitle);

metadataBuilder.putString(MediaMetadata.METADATA_KEY_DISPLAY_SUBTITLE,

            myData.displaySubtitle);

metadataBuilder.putString(MediaMetadata.METADATA_KEY_DISPLAY_ICON_URI,

            myData.artUri);
    // And at minimum the title and artist for legacy support
```

```
    metadataBuilder.putString(MediaMetadata.METADATA_KEY_TITLE,
            myData.title);
    metadataBuilder.putString(MediaMetadata.METADATA_KEY_ARTIST,
            myData.artist);
    // A small bitmap for the artwork is also recommended
    metadataBuilder.putString(MediaMetadata.METADATA_KEY_ART,
            myData.artBitmap);
    // Add any other fields you have for your data as well
    mSession.setMetadata(metadataBuilder.build());
}
```

Respond to User Action

When the user selects the Now Playing card, the system opens the app that owns the session. If your app provides a `PendingIntent` to pass to `setSessionActivity()`, the system launches the activity you specify, as demonstrated below. If not, the default system intent opens. The activity you specify must provide playback controls that allow users to pause or stop playback.

```
Intent intent = new Intent(mContext, MyActivity.class);
    PendingIntent pi = PendingIntent.getActivity(context, 99
/*request code*/,
            intent, PendingIntent.FLAG_UPDATE_CURRENT);
    mSession.setSessionActivity(pi);
```

Next Get started Next class

Helping Users Find Your Content on TV

Previous Next Get started

Dependencies and Prerequisites

- Android 5.0 (API level 21) or higher

You should also read

- Design for TV

TV devices offer many entertainment options for users. They have thousands of content options from apps and related content services. At the same time, most users prefer to use TVs with the least amount of input possible. With the number of choices available to users, it is important for app developers to provide quick and easy paths for users to discover and enjoy your content.

The Android framework helps you provide a number of paths for users to discover your content, including recommendations on the home screen and searching within your app's content catalog.

This class shows you how to help users discover your app's content through recommendations and in-app searching.

Topics

Recommending TV Content

Learn how to recommend content for users so that it appears in the recommendations row on the home screen of a TV device.

Making TV Apps Searchable

Learn how to make your content searchable from the Android TV home screen.

Searching within TV Apps

Learn how to use a built-for-TV user interface for searching within your app.

Next Get started Next class

Recommending TV Content

Previous Next Get started

This lesson teaches you to

1. Best Practices for Recommendations
2. Create a Recommendations Service
3. Build Recommendations
4. Run Recommendations Service

Try it out

- Android Leanback sample app

When interacting with TVs, users generally prefer to give minimal input before watching content. An ideal scenario for many TV users is: sit down, turn on, and watch. The fewest steps to get users to content they enjoy is generally the path they prefer.

The Android framework assists with minimum-input interaction by providing a recommendations row on the home screen. Content recommendations appear as the first row of the TV home screen after the first use of the device. Contributing recommendations from your app's content catalog can help bring users back to your app.

Figure 1. An example of the recommendations row.

This lesson teaches you how to create recommendations and provide them to the Android framework so users can easily discover and enjoy your app content. This discussion describes some code from the Android Leanback sample app.

Best Practices for Recommendations

Recommendations help users quickly find the content and apps they enjoy. Creating recommendations that are high-quality and relevant to users is an important factor in creating a great user experience with your TV app. For this reason, you should carefully consider what recommendations you present to the user and manage them closely.

Types of Recommendations

When you create recommendations, you should link users back to incomplete viewing activities or suggest activities that extend that to related content. Here are some specific type of recommendations you should consider:

- **Continuation content** recommendations for the next episode for users to resume watching a series.
- **New content** recommendations, such as for a new first-run episode, if the user finished watching another series.
- **Related content** recommendations based on the users historic viewing behavior.

For more information on how to design recommendation cards for the best user experience, see Recommendation Row in the Android TV Design Spec.

Refreshing Recommendations

When refreshing recommendations, don't just remove and repost them, because doing so causes the recommendations to appear at the end of the recommendations row. Once a content item, such as a movie, has been played, remove it from the recommendations.

Customizing Recommendations

You can customize recommendation cards to convey branding information, by setting user interface elements such as the card's foreground and background image, color, app icon, title, and subtitle. To learn more, see Recommendation Row in the Android TV Design Spec.

Create a Recommendations Service

Content recommendations are created with background processing. In order for your application to contribute to recommendations, create a service that periodically adds listings from your app's catalog to the system's list of recommendations.

The following code example illustrates how to extend `IntentService` to create a recommendation service for your application:

```
public class UpdateRecommendationsService extends IntentService {
    private static final String TAG = "UpdateRecommendationsService";
    private static final int MAX_RECOMMENDATIONS = 3;

    public UpdateRecommendationsService() {
        super("RecommendationService");
    }

    @Override
    protected void onHandleIntent(Intent intent) {
        Log.d(TAG, "Updating recommendation cards");
        HashMap<String, List<Movie>> recommendations = VideoProvider.getMovieList();
        if (recommendations == null) return;

        int count = 0;

        try {
            RecommendationBuilder builder = new RecommendationBuilder()
                    .setContext(getApplicationContext())
                    .setSmallIcon(R.drawable.videos_by_google_icon);

            for (Map.Entry<String, List<Movie>> entry : recommendations.entrySet()) {
                for (Movie movie : entry.getValue()) {
                    Log.d(TAG, "Recommendation - " + movie.getTitle());

                    builder.setBackground(movie.getCardImageUrl())
                            .setId(count + 1)
                            .setPriority(MAX_RECOMMENDATIONS - count)
                            .setTitle(movie.getTitle())
```

```
                            .setDescription(getString(R.string.popular_header))
                            .setImage(movie.getCardImageUrl())
                            .setIntent(buildPendingIntent(movie))
                            .build();

                    if (++count >= MAX_RECOMMENDATIONS) {
                        break;
                    }
                }
                if (++count >= MAX_RECOMMENDATIONS) {
                    break;
                }
            }
        } catch (IOException e) {
            Log.e(TAG, "Unable to update recommendation", e);
        }
    }

    private PendingIntent buildPendingIntent(Movie movie) {
        Intent detailsIntent = new Intent(this, DetailsActivity.class);
        detailsIntent.putExtra("Movie", movie);

        TaskStackBuilder stackBuilder = TaskStackBuilder.create(this);
        stackBuilder.addParentStack(DetailsActivity.class);
        stackBuilder.addNextIntent(detailsIntent);
        // Ensure a unique PendingIntents, otherwise all
        // recommendations end up with the same PendingIntent
        detailsIntent.setAction(Long.toString(movie.getId()));

        PendingIntent intent = stackBuilder.getPendingIntent(0, PendingIntent.FLAG_UPDATE_CURRENT);
        return intent;
    }
}
```

In order for this service to be recognized by the system and run, register it using your app manifest. The following code snippet illustrates how to declare this class as a service:

```
<manifest ... >
  <application ... >
    ...

    <service
```

```
            android:name="com.example.android.tvleanback.UpdateRecommendationsService"
            android:enabled="true" />
  </application>
</manifest>
```

Build Recommendations

Once your recommendation service starts running, it must create recommendations and pass them to the Android framework. The framework receives the recommendations as `Notification` objects that use a specific template and are marked with a specific category.

Setting the Values

To set the UI element values for the recommendation card, you create a builder class that follows the builder pattern described as follows. First, you set the values of the recommendation card elements.

```
public class RecommendationBuilder {
    ...

    public RecommendationBuilder setTitle(String title) {
            mTitle = title;
            return this;
        }

        public RecommendationBuilder setDescription(String description) {
            mDescription = description;
            return this;
        }

        public RecommendationBuilder setImage(String uri) {
            mImageUri = uri;
            return this;
        }

        public RecommendationBuilder setBackground(String uri) {
            mBackgroundUri = uri;
```

```
            return this;
        }
...
```

Creating the Notification

Once you've set the values, you then build the notification, assigning the values from the builder class to the notification, and calling `NotificationCompat.Builder.build()`.

Also, be sure to call `setLocalOnly()` so the `NotificationCompat.BigPictureStyle` notification won't show up on other devices.

The following code example demonstrates how to build a recommendation.

```
public class RecommendationBuilder {
    ...

    public Notification build() throws IOException {
        ...

        Notification notification = new NotificationCompat.BigPictureStyle(
                new NotificationCompat.Builder(mContext)
                        .setContentTitle(mTitle)
                        .setContentText(mDescription)
                        .setPriority(mPriority)
                        .setLocalOnly(true)
                        .setOngoing(true)
                        .setColor(mContext.getResources().getColor(R.color.fastlane_background))
                        .setCategory(Notification.CATEGORY_RECOMMENDATION)
                        .setLargeIcon(image)
                        .setSmallIcon(mSmallIcon)
                        .setContentIntent(mIntent)
                        .setExtras(extras))
                .build();

        return notification;
    }
}
```

Run Recommendations Service

Your app's recommendation service must run periodically in order to create current recommendations. To run your service, create a class that runs a timer and invokes it at regular intervals. The following code example extends the `BroadcastReceiver` class to start periodic execution of a recommendation service every half hour:

```
public class BootupActivity extends BroadcastReceiver {
    private static final String TAG = "BootupActivity";

    private static final long INITIAL_DELAY = 5000;

    @Override
    public void onReceive(Context context, Intent intent) {
        Log.d(TAG, "BootupActivity initiated");
        if
(intent.getAction().endsWith(Intent.ACTION_BOOT_COMPLETED)) {
            scheduleRecommendationUpdate(context);
        }
    }
```

```
    private void scheduleRecommendationUpdate(Context context) {
        Log.d(TAG, "Scheduling recommendations update");

        AlarmManager alarmManager = (AlarmManager)
context.getSystemService(Context.ALARM_SERVICE);
        Intent recommendationIntent = new Intent(context,
UpdateRecommendationsService.class);
        PendingIntent alarmIntent =
PendingIntent.getService(context, 0, recommendationIntent, 0);


alarmManager.setInexactRepeating(AlarmManager.ELAPSED_REALTIME_WAKEU

                INITIAL_DELAY,
                AlarmManager.INTERVAL_HALF_HOUR,
                alarmIntent);
    }
}
```

This implementation of the `BroadcastReceiver` class must run after start up of the TV device where it is installed. To accomplish this, register this class in your app manifest with an intent filter that listens for the completion of the device boot process. The following sample code demonstrates how to add this

configuration to the manifest:

```
<manifest ... >
  <application ... >
    <receiver
android:name="com.example.android.tvleanback.BootupActivity"
              android:enabled="true"
              android:exported="false">
      <intent-filter>
        <action
android:name="android.intent.action.BOOT_COMPLETED"/>
      </intent-filter>
    </receiver>
  </application>
</manifest>
```

Important: Receiving a boot completed notification requires that your app requests the `RECEIVE_BOOT_COMPLETED` permission. For more information, see `ACTION_BOOT_COMPLETED`.

In your recommendation service class' `onHandleIntent()` method, post the recommendation to the manager as follows:

```
Notification notification = notificationBuilder.build();
mNotificationManager.notify(id, notification);
```

Next Get started Next class

Making TV Apps Searchable

Previous Next Get started

This lesson teaches you to

1. Identify Columns
2. Provide Search Suggestion Data
3. Handle Search Suggestions
4. Handle Search Terms
5. Deep Link to Your App in the Details Screen

You should also read

- Search
- Adding Search Functionality

Try it out

- Android Leanback sample app

Android TV uses the Android search interface to retrieve content data from installed apps and deliver search results to the user. Your app's content data can be included with these results, to give the user instant access to the content in your app.

Your app must provide Android TV with the data fields from which it generates suggested search results as the user enters characters in the search dialog. To do that, your app must implement a Content Provider that serves up the suggestions along with a `searchable.xml` configuration file that describes the content provider and other vital information for Android TV. You also need an activity that handles the intent that fires when the user selects a suggested search result. All of this is described in

more detail in Adding Custom Suggestions. Here are described the main points for Android TV apps.

This lesson builds on your knowledge of using search in Android to show you how to make your app searchable in Android TV. Be sure you are familiar with the concepts explained in the Search API guide before following this lesson. See also the training Adding Search Functionality.

This discussion describes some code from the Android Leanback sample app, available on GitHub.

Identify Columns

The `SearchManager` describes the data fields it expects by representing them as columns of an SQLite database. Regardless of your data's format, you must map your data fields to these columns, usually in the class that accessess your content data. For information about building a class that maps your existing data to the required fields, see Building a suggestion table.

The `SearchManager` class includes several columns for Android TV. Some of the more important columns are described below.

Value	Description
`SUGGEST_COLUMN_TEXT_1`	The name of your content **(required)**
`SUGGEST_COLUMN_TEXT_2`	A text description of your content
`SUGGEST_COLUMN_RESULT_CARD_IMAGE`	An image/poster/cover for your content
`SUGGEST_COLUMN_CONTENT_TYPE`	The MIME type of your media **(required)**
`SUGGEST_COLUMN_VIDEO_WIDTH`	The resolution width of your media
`SUGGEST_COLUMN_VIDEO_HEIGHT`	The resolution height of your media
`SUGGEST_COLUMN_PRODUCTION_YEAR`	The production year of your content **(required)**
`SUGGEST_COLUMN_DURATION`	The duration in milliseconds of your media **(required)**

The search framework requires the following columns:

- SUGGEST_COLUMN_TEXT_1
- SUGGEST_COLUMN_CONTENT_TYPE
- SUGGEST_COLUMN_PRODUCTION_YEAR
- SUGGEST_COLUMN_DURATION

When the values of these columns for your content match the values for the same content from other providers found by Google servers, the system provides a deep link to your app in the details view for the content, along with links to the apps of other providers. This is discussed more in Display Content in the Details Screen, below.

Your application's database class might define the columns as follows:

```
public class VideoDatabase {
  //The columns we'll include in the video database table
  public static final String KEY_NAME =
SearchManager.SUGGEST_COLUMN_TEXT_1;
  public static final String KEY_DESCRIPTION =
SearchManager.SUGGEST_COLUMN_TEXT_2;
  public static final String KEY_ICON =
SearchManager.SUGGEST_COLUMN_RESULT_CARD_IMAGE;
  public static final String KEY_DATA_TYPE =
SearchManager.SUGGEST_COLUMN_CONTENT_TYPE;
  public static final String KEY_IS_LIVE =
SearchManager.SUGGEST_COLUMN_IS_LIVE;
  public static final String KEY_VIDEO_WIDTH =
SearchManager.SUGGEST_COLUMN_VIDEO_WIDTH;
  public static final String KEY_VIDEO_HEIGHT =
SearchManager.SUGGEST_COLUMN_VIDEO_HEIGHT;
  public static final String KEY_AUDIO_CHANNEL_CONFIG =
           SearchManager.SUGGEST_COLUMN_AUDIO_CHANNEL_CONFIG;
  public static final String KEY_PURCHASE_PRICE =
SearchManager.SUGGEST_COLUMN_PURCHASE_PRICE;
  public static final String KEY_RENTAL_PRICE =
SearchManager.SUGGEST_COLUMN_RENTAL_PRICE;
  public static final String KEY_RATING_STYLE =
SearchManager.SUGGEST_COLUMN_RATING_STYLE;
  public static final String KEY_RATING_SCORE =
SearchManager.SUGGEST_COLUMN_RATING_SCORE;
  public static final String KEY_PRODUCTION_YEAR =
SearchManager.SUGGEST_COLUMN_PRODUCTION_YEAR;
  public static final String KEY_COLUMN_DURATION =
SearchManager.SUGGEST_COLUMN_DURATION;
  public static final String KEY_ACTION =
SearchManager.SUGGEST_COLUMN_INTENT_ACTION;
...
```

When you build the map from the `SearchManager` columns to your data fields, you must also specify the `_ID` to give each row a unique ID.

```
...
  private static HashMap buildColumnMap() {
    HashMap map = new HashMap();
    map.put(KEY_NAME, KEY_NAME);
    map.put(KEY_DESCRIPTION, KEY_DESCRIPTION);
    map.put(KEY_ICON, KEY_ICON);
    map.put(KEY_DATA_TYPE, KEY_DATA_TYPE);
    map.put(KEY_IS_LIVE, KEY_IS_LIVE);
    map.put(KEY_VIDEO_WIDTH, KEY_VIDEO_WIDTH);
    map.put(KEY_VIDEO_HEIGHT, KEY_VIDEO_HEIGHT);
    map.put(KEY_AUDIO_CHANNEL_CONFIG, KEY_AUDIO_CHANNEL_CONFIG);
    map.put(KEY_PURCHASE_PRICE, KEY_PURCHASE_PRICE);
    map.put(KEY_RENTAL_PRICE, KEY_RENTAL_PRICE);
    map.put(KEY_RATING_STYLE, KEY_RATING_STYLE);
    map.put(KEY_RATING_SCORE, KEY_RATING_SCORE);
    map.put(KEY_PRODUCTION_YEAR, KEY_PRODUCTION_YEAR);
    map.put(KEY_COLUMN_DURATION, KEY_COLUMN_DURATION);
    map.put(KEY_ACTION, KEY_ACTION);
    map.put(BaseColumns._ID, "rowid AS " +
            BaseColumns._ID);
    map.put(SearchManager.SUGGEST_COLUMN_INTENT_DATA_ID, "rowid AS "
+
            SearchManager.SUGGEST_COLUMN_INTENT_DATA_ID);
    map.put(SearchManager.SUGGEST_COLUMN_SHORTCUT_ID, "rowid AS " +
            SearchManager.SUGGEST_COLUMN_SHORTCUT_ID);
    return map;
  }
...
```

In the example above, notice the mapping to the `SUGGEST_COLUMN_INTENT_DATA_ID` field. This is the portion of the URI that points to the content unique to the data in this row — that is, the last part of the URI describing where the content is stored. The first part of the URI, when it is common to all of the rows in the table, is set in the `searchable.xml` file as the `android:searchSuggestIntentData` attribute, as described in Handle Search Suggestions, below.

If the first part of the URI is different for each row in the table, you map that value with the `SUGGEST_COLUMN_INTENT_DATA` field. When the user selects this content, the intent that fires provides the intent data from the combination of the `SUGGEST_COLUMN_INTENT_DATA_ID` and either the `android:searchSuggestIntentData` attribute or the `SUGGEST_COLUMN_INTENT_DATA` field value.

Provide Search Suggestion Data

Implement a Content Provider to return search term suggestions to the Android TV search dialog. The system queries your content provider for

suggestions by calling the `query()` method each time a letter is typed. In your implementation of `query()`, your content provider searches your suggestion data and returns a `Cursor` that points to the rows you have designated for suggestions.

```
@Override
  public Cursor query(Uri uri, String[] projection, String selection,
String[] selectionArgs,
                      String sortOrder) {
    // Use the UriMatcher to see what kind of query we have and
format the db query accordingly
    switch (URI_MATCHER.match(uri)) {
      case SEARCH_SUGGEST:
          Log.d(TAG, "search suggest: " + selectionArgs[0] + " URI: "
+ uri);
          if (selectionArgs == null) {
              throw new IllegalArgumentException(
                      "selectionArgs must be provided for the Uri: "
+ uri);
          }
          return getSuggestions(selectionArgs[0]);
      default:
          throw new IllegalArgumentException("Unknown Uri: " + uri);
    }
  }

  private Cursor getSuggestions(String query) {
    query = query.toLowerCase();
    String[] columns = new String[]{
      BaseColumns._ID,
      VideoDatabase.KEY_NAME,
      VideoDatabase.KEY_DESCRIPTION,
      VideoDatabase.KEY_ICON,
      VideoDatabase.KEY_DATA_TYPE,
      VideoDatabase.KEY_IS_LIVE,
      VideoDatabase.KEY_VIDEO_WIDTH,
      VideoDatabase.KEY_VIDEO_HEIGHT,
      VideoDatabase.KEY_AUDIO_CHANNEL_CONFIG,
      VideoDatabase.KEY_PURCHASE_PRICE,
      VideoDatabase.KEY_RENTAL_PRICE,
      VideoDatabase.KEY_RATING_STYLE,
      VideoDatabase.KEY_RATING_SCORE,
      VideoDatabase.KEY_PRODUCTION_YEAR,
      VideoDatabase.KEY_COLUMN_DURATION,
      VideoDatabase.KEY_ACTION,
      SearchManager.SUGGEST_COLUMN_INTENT_DATA_ID
    };
    return mVideoDatabase.getWordMatch(query, columns);
  }
...
```

In your manifest file, the content provider receives special treatment. Rather than getting tagged as an activity, it is described as a `<provider>`. The provider includes the `android:searchSuggestAuthority` attribute to tell the system the namespace of your content provider. Also, you must set its

`android:exported` attribute to `"true"` so that the Android global search can use the results returned from it.

```
<provider
android:name="com.example.android.tvleanback.VideoContentProvider"
    android:authorities="com.example.android.tvleanback"
    android:exported="true" />
```

Handle Search Suggestions

Your app must include a `res/xml/searchable.xml` file to configure the search suggestions settings. It inlcudes the `android:searchSuggestAuthority` attribute to tell the system the namespace of your content provider. This must match the string value you specify in the `android:authorities` attribute of the `<provider>` element in your `AndroidManifest.xml` file.

The `searchable.xml` file must also include the `android:searchSuggestIntentAction` with the value `"android.intent.action.VIEW"` to define the intent action for providing a custom suggestion. This is different from the intent action for providing a search term, explained below. See also, Declaring the intent action for other ways to declare the intent action for suggestions.

Along with the intent action, your app must provide the intent data, which you specify with the `android:searchSuggestIntentData` attribute. This is the first part of the URI that points to the content. It describes the portion of the URI common to all rows in the mapping table for that content. The portion of the URI that is unique to each row is established with the `SUGGEST_COLUMN_INTENT_DATA_ID` field, as described above in Identify Columns. See also, Declaring the intent data for other ways to declare the intent data for suggestions.

Also, note the `android:searchSuggestSelection=" ?"` attribute which specifies the value passed as the `selection` parameter of the `query()` method where the question mark (`?`) value is replaced with the query text.

Finally, you must also include the `android:includeInGlobalSearch` attribute

with the value `"true"`. Here is an example `searchable.xml` file:

```
<searchable xmlns:android="http://schemas.android.com/apk/res/android"
    android:label="@string/search_label"
        android:hint="@string/search_hint"

android:searchSettingsDescription="@string/settings_description"

android:searchSuggestAuthority="com.example.android.tvleanback"
        android:searchSuggestIntentAction="android.intent.action.VIEW"

android:searchSuggestIntentData="content://com.example.android.tvleanba

        android:searchSuggestSelection=" ?"
        android:searchSuggestThreshold="1"
        android:includeInGlobalSearch="true"
    >
</searchable>
```

Handle Search Terms

As soon as the search dialog has a word which matches the value in one of your app's columns (described in Identifying Columns, above), the system fires the `ACTION_SEARCH` intent. The activity in your app which handles that intent searches the repository for columns with the given word in their values, and returns a list of content items with those columns. In your `AndroidManifest.xml` file, you designate the activity which handles the `ACTION_SEARCH` intent like this:

```
...
  <activity
      android:name="com.example.android.tvleanback.DetailsActivity"
      android:exported="true">

      <!-- Receives the search request. -->
      <intent-filter>
          <action android:name="android.intent.action.SEARCH" />
          <!-- No category needed, because the Intent will specify
this class component -->
      </intent-filter>

      <!-- Points to searchable meta data. -->
      <meta-data android:name="android.app.searchable"
          android:resource="@xml/searchable" />
  </activity>
...
  <!-- Provides search suggestions for keywords against video meta
data. -->
  <provider
android:name="com.example.android.tvleanback.VideoContentProvider"
      android:authorities="com.example.android.tvleanback"
      android:exported="true" />
...
```

The activity must also describe the searchable configuration with a reference to the `searchable.xml` file. To use the global search dialog, the manifest must describe which activity should receive search queries. The manifest must also describe the `<provider>` element, exactly as it is described in the `searchable.xml` file.

Deep Link to Your App in the Details Screen

If you have set up the search configuration as described in Handle Search Suggestions and mapped the `SUGGEST_COLUMN_TEXT_1`, `SUGGEST_COLUMN_CONTENT_TYPE`, and `SUGGEST_COLUMN_PRODUCTION_YEAR` fields as described in Identify Columns, a deep link to a watch action for your content appears in the details screen that launches when the user selects a search result, as shown in figure 1.

Figure 1. The details screen displays a deep link for the Videos by Google (Leanback) sample app. Sintel: © copyright Blender Foundation, www.sintel.org.

When the user selects the link for your app, identified by the "Available

On" button in the details screen, the system launches the activity which handles the `ACTION_VIEW` (set as `android:searchSuggestIntentAction` with the value `"android.intent.action.VIEW"` in the `searchable.xml` file).

You can also set up a custom intent to launch your activity, and this is demonstrated in the Android Leanback sample app. Note that the sample app launches its own `LeanbackDetailsFragment` to show the details for the selected media, but you should launch the activity that plays the media immediately to save the user another click or two.

Next Get started Next class

Searching within TV Apps

Previous Next Get started

This lesson teaches you to

1. Add a Search Action
2. Add Search Input and Results

Users frequently have specific content in mind when using a media app on TV. If your app contains a large catalog of content, browsing for a specific title may not be the most efficient way for users to find what they are looking for. A search interface can help your users get to the content they want faster than browsing.

The Leanback support library provides a set of classes to enable a standard search interface within your app that is consistent with other search functions on TV and provides features such as voice input.

This lesson discusses how to provide a search interface in your app using Leanback support library classes.

Add a Search Action

When you use the `BrowseFragment` class for a media browsing interface, you can enable a search interface as a standard part of the user interface. The search interface is an icon that appears in the layout when you set `View.OnClickListener` on the `BrowseFragment` object. The following sample code demonstrates this technique.

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.browse_activity);

    mBrowseFragment = (BrowseFragment)
```

```
getFragmentManager().findFragmentById(R.id.browse_fragment);

    ...

    mBrowseFragment.setOnSearchClickedListener(new
View.OnClickListener() {
        @Override
        public void onClick(View view) {
            Intent intent = new Intent(BrowseActivity.this,
SearchActivity.class);
            startActivity(intent);
        }
    });

    mBrowseFragment.setAdapter(buildAdapter());
}
```

Note: You can set the color of the search icon using the `setSearchAffordanceColor(int)`.

Add a Search Input and Results

When a user selects the search icon, the system invokes a search activity via the defined intent. Your search activity should use a linear layout containing a `SearchFragment`. This fragment must also implement the `SearchFragment.SearchResultProvider` interface in order to display the results of a search.

The following code sample shows how to extend the `SearchFragment` class to provide a search interface and results:

```
public class MySearchFragment extends SearchFragment
        implements SearchFragment.SearchResultProvider {

    private static final int SEARCH_DELAY_MS = 300;
    private ArrayObjectAdapter mRowsAdapter;
    private Handler mHandler = new Handler();
    private SearchRunnable mDelayedLoad;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        mRowsAdapter = new ArrayObjectAdapter(new
ListRowPresenter());
        setSearchResultProvider(this);
        setOnItemClickedListener(getDefaultItemClickedListener());
        mDelayedLoad = new SearchRunnable();
    }

    @Override
```

```
    public ObjectAdapter getResultsAdapter() {
        return mRowsAdapter;
    }

    @Override
    public boolean onQueryTextChange(String newQuery) {
        mRowsAdapter.clear();
        if (!TextUtils.isEmpty(newQuery)) {
            mDelayedLoad.setSearchQuery(newQuery);
            mHandler.removeCallbacks(mDelayedLoad);
            mHandler.postDelayed(mDelayedLoad, SEARCH_DELAY_MS);
        }
        return true;
    }

    @Override
    public boolean onQueryTextSubmit(String query) {
        mRowsAdapter.clear();
        if (!TextUtils.isEmpty(query)) {
            mDelayedLoad.setSearchQuery(query);
            mHandler.removeCallbacks(mDelayedLoad);
            mHandler.postDelayed(mDelayedLoad, SEARCH_DELAY_MS);
        }
        return true;
    }
}
```

The example code shown above is meant to be used with a separate `SearchRunnable` class that runs the search query on a separate thread. This technique keeps potentially slow-running queries from blocking the main user interface thread.

Next Get started Next class

Building TV Games

In this document

1. Display
2. Input Devices
3. Manifest
4. Google Play Game Services
5. Web

The television screen presents a number of considerations that may be new to mobile game developers. These areas include its large size, its control scheme, and the fact that all players are viewing it simultaneously.

Display

The two main things to keep in mind when developing games for the TV screen are its nature as a shared display and the need to design your game for a landscape orientation.

Consider the shared display

A living-room TV poses design challenges for multiplayer games, in that all players can see everything. This issue is especially relevant to games (such as card games or strategy games) that rely on each player's possession of hidden information.

Some mechanisms you can implement to address the problem of one player's eavesdropping on another's information are:

- A blinder on the screen to help conceal information. For example, in a turn-based game like a word or card game, one player at a time might view the display. When the player finishes a move, the game allows him or her to cover the screen with a blinder that blocks anyone from viewing secret information. When the next player begins a turn, the blinder opens to reveal his or her own information.
- A companion app, running on a phone or tablet, can enable a player to

conceal information by serving as a second screen.

Support landscape display

A TV is always sideways: You can't turn it, and there is no portrait orientation. Always design your TV games to be displayed in landscape mode.

Input Devices

TVs don't have touch interfaces, so it's even more important to get your controls right and make sure players find them intuitive and fun to use. Handling controllers also introduces some other issues to pay attention to, like keeping track of multiple controllers, and handling disconnects gracefully.

Support D-pad controls

Plan your control scheme around a directional pad (D-pad) control, since this control set is the default for Android TV devices. The player needs to be able to use a D-Pad in all aspects of the game—not just controlling core gameplay, but also navigating menus and ads. For this reason, you should also ensure that your Android TV game does not refer to a touch interface. For example, an Android TV game should not tell a player to "*Tap* here to continue."

How you shape the player's interaction with the controller can be key to achieving a great user experience:

- **Communicate Controller Requirements up Front**. Use your Google Play description to communicate to the player any expectations about controllers. If a game is better suited to a gamepad with a joystick than one with only a D-pad, make this fact clear. A player who uses an ill-suited controller for a game is likely to have a subpar experience and penalize your game in the ratings.
- **Use Consistent Button Mapping**. Intuitive and flexible button mapping is key to a good user experience. For example, you should adhere to accepted customs by using the A button to *Accept*, and the B button to *Cancel*. You can also offer flexibility in the form of remappability. For more information about button mapping, see Handling Controller Actions.
- **Detect Controller Capabilities and Adjust Accordingly**. Query the controller about its capabilities in order to optimize the match between controller and game. For example, you may intend for a player to steer an object by waving the controller in the air. If a player's controller lacks

accelerometer and gyroscope hardware, however, waving will not work. So, your game should query the controller and if motion detection is not supported, switch over to an alternative, available control scheme. For more information about querying controller capabilities, see Supporting Controllers Across Android Versions.

Provide appropriate Back-button behavior

The Back button should never act as a toggle. For example, do not use it to both open and close a menu. It should only navigate backward, breadcrumb-style, through the previous screens the player has been on, for example: Game play > Game pause screen > Game main screen > Android home screen.

Since the Back button should only perform linear (backward) navigation, you may use the back button to leave an in-game menu (opened by a different button) and return to gameplay. For more information about design for navigation, see Navigation with Back and Up. To learn about implementation, refer to Providing Proper Back Navigation.

Use appropriate buttons

Not all game controllers provide Start, Search, or Menu buttons. Be sure your UI does not depend upon the use of these buttons.

Handle multiple controllers

When multiple players are playing a game, each with his or her own controller, it is important to map each player-controller pair. For information about how to implement controller-number identification, see Input Devices.

Handle controller disconnects

When a controller is disconnected in the middle of gameplay, the game should pause, and a dialog should appear prompting the disconnected player to reconnect his or her controller.

The dialog should also offer troubleshooting tips (for example, a pop-up dialog telling the player to "Check your Bluetooth connection"). For more information about implementing input-device support, see Handling Controller Actions. Specific information about Bluetooth connections is at Bluetooth.

Show controller instructions

If your game provides visual game control instructions, the controller image should be free of branding and include only buttons compatible with Android.

For sample images of an Android-compatible controller, download the Android TV Gamepad Template (ZIP). It includes a white controller on black background and a black controller on white background (shown in figure 1), as a PNG file and an Adobe® Illustrator® file.

Figure 1. Example controller instructions using the Android TV Gamepad Template (ZIP).

Manifest

There are a some special things games should include in the Android manifest.

Show your game on the home screen

The Android TV home screen displays games in a separate row from regular apps. To make your game appear in the list of games, set the `android:isGame` attribute to `"true"` in your app manifest's `<application>` tag. For example:

```
<application
    ...
    android:isGame="true"
    ...
>
```

Declare support for game controllers

Games controllers may not be available or active for users of a TV device. In order to properly inform users that your game requires (or just supports) a game controller, you must include entries in the app manifest. If your game requires a

game controller, you must include the following entry in your app manifest:

```
<uses-feature android:name="android.hardware.gamepad"/>
```

If your game uses, but does not require, a game controller, include the following feature entry in your app manifest:

```
<uses-feature android:name="android.hardware.gamepad"
android:required="false"/>
```

For more information about manifest entries, see App Manifest.

Google Play Game Services

If your game integrates Google Play Game services, you should keep in mind a number of considerations pertaining to achievements, sign-in, saving games, and multiplayer play.

Achievements

Your game should include at least five (earnable) achievements. Only a user controlling gameplay from a supported input device should be able to earn achievements. For more information about achievements and how to implement them, see Achievements in Android.

Sign-in

Your game should attempt to sign the user in on launch. If the player declines sign-in several times in a row, your game should stop asking. Learn more about sign-in at Implementing Sign-in on Android.

Saving

Use Google Play Services Saved Games to store your game save. Your game should bind game saves to a specific Google account, so as to be uniquely identifiable even across devices: Whether the player is using a handset or a TV, the game should be able to pull the game-save information from the same user account.

You should also provide an option in your game's UI to allow the player to delete locally and cloud-stored data. You might put the option in the game's `Settings` screen. For specifics on implementing saved games using Play Services, see Saved Games in Android.

Multiplayer experience

A game offering a multiplayer experience must allow at least two players to enter a room. For further information about multiplayer games in Android, see the Real-time Multiplayer and Turn-based Multiplayer documentation on the Android developer site.

Exit

Provide a consistent and obvious UI element that lets the user exit the game gracefully. This element should be accessible with the D-pad navigation buttons. Do this instead of relying on the Home button to provide an exit, as that is not consistent nor reliable across different controllers.

Web

Do not enable web browsing in games for Android TV. Android TV does not support a web browser.

Note: You can use the `WebView` class for logins to services like Google+ and Facebook.

Networking

Games frequently need greater bandwidth to provide optimum performance, and many users prefer ethernet to WiFi to provide that performance. Your app should check for both WiFi and ethernet connections. If your app is for TV only, you do not need to check for 3G/LTE service as you would for a mobile app.

Next Get started Next class

Building TV Channels

Previous Next Get started

Dependencies and Prerequisites

- Android 5.0 (API level 21) or higher

You should also read

- android.media.tv

Try It Out

- TV Input Service sample app

Watching live TV shows and other continuous, channel-based content is a big part of the TV experience. Users are accustomed to selecting and watching shows on TV by channel browsing. To provide your users a similar experience, use the TV Input Framework to create channels for publishing video or music content so that your media appears alongside traditional TV channels in the programming guide.

Android supports receiving and playback of live video content through the TV Input Framework in Android 5.0 (API level 21). This framework provides a unified method for receiving audio and video channel content from hardware sources, such as HDMI ports and built-in-tuners, and software sources, such as video streamed over the internet.

The framework enables developers to define live TV input sources by implementing a TV input service. This service publishes a list of channels and programs to the TV Provider. The live TV app on a TV device gets the list of available channels and programs from the TV Provider and displays them to a user. When a user selects a specific channel, the live TV app creates a session for the associated TV input service through the TV Input Manager, and tells the TV input service to tune to the requested channel and play the content to a display surface provided by the TV app.

Figure 1. Functional diagram of the TV Input Framework

The TV Input Framework is designed to provide access to a wide variety of live TV input sources and bring them together in a single user interface for users to browse, view, and enjoy content. Building a TV input service for your content can help make it more accessible on TV devices.

Topics

Developing a TV Input Service
Learn how to develop a TV input service, which works with the system TV app.

Working with Channel Data
Learn how to describe channel and program data for the system.

Managing User Interaction
Learn how to present overlays, manage content availability, and handle content selection.

Next Get started Next class

Developing a TV Input Service

Previous Next Get started

This lesson teaches you to

1. Declare Your TV Input Service in the Manifest
2. Define Your TV Input Service
3. Define Setup and Settings Activities

You should also read

- android.media.tv
- TV Input Framework

Try It Out

- TV Input Service sample app

A TV input service represents a media stream source, and lets you present your media content in a linear, broadcast TV fashion as channels and programs. With the TV input service, you can provide parental controls, program guide information, and content ratings. The TV input service works with the Android system TV app, developed for the device and immutable by third-party apps, which ultimately controls and presents content on the TV. See TV Input Framework for more information about the framework architecture and its components.

To develop a TV input service, you implement the following components:

- `TvInputService` provides long-running and background availability for the TV input
- `TvInputService.Session` maintains the TV input state and

communicates with the hosting app

- `TvContract` describes the channels and programs available to the TV input
- `TvContract.Channels` represents information about a TV channel
- `TvContract.Programs` describes a TV program with data such as program title and start time
- `TvTrackInfo` represents an audio, video, or subtitle track
- `TvContentRating` describes a content rating, allows for custom content rating schemes
- `TvInputManager` provides an API to the system TV app and manages the interaction with TV inputs and apps

Declare Your TV Input Service in the Manifest

Your app manifest must declare your `TvInputService`. Within that declaration, specify the `BIND_TV_INPUT` permission to allow the service to connect the TV input to the system. A system service (`TvInputManagerService`) performs the binding and has that permission. The system TV app sends requests to TV input services via the `TvInputManager` interface. The service declaration must also include an intent filter that specifies the `TvInputService` as the action to perform with the intent. Also within the service declaration, declare the service meta data in a separate XML resource. The service declaration, the intent filter and the service meta data are described in the following example.

```
<service android:name="com.example.sampletvinput.SampleTvInput"
    android:label="@string/sample_tv_input_label"
    android:permission="android.permission.BIND_TV_INPUT">
    <intent-filter>
        <action android:name="android.media.tv.TvInputService" />
    </intent-filter>
    <meta-data android:name="android.media.tv.input"
      android:resource="@xml/sample_tv_input" />
</service>
```

Define the service meta data in separate XML file, as shown in the following example. The service meta data must include a setup interface that describes the TV input's initial configuration and channel scan. Also, the service meta data may (optionally) describe a settings activity for

users to modify the TV input's behavior. The service meta data file is located in the XML resources directory for your application and must match the name of the resource in the manifest. Using the example manifest entries above, you would create an XML file in the location `res/xml/sample_tv_input.xml`, with the following contents:

```
<tv-input xmlns:android="http://schemas.android.com/apk/res/android"
  <!-- Required: activity for setting up the input -->

android:setupActivity="com.example.sampletvinput.SampleTvInputSetupActi

  <!-- Optional: activity for controlling the settings -->

android:settingsActivity="com.example.sampletvinput.SampleTvInputSettin
/>
```

Define Your TV Input Service

Figure 1. TvInputService lifecycle.

For your service, you extend the `TvInputService` class. A `TvInputService` implementation is a bound service where the system service (`TvInputManagerService`) is the client that binds to it. The service life cycle methods you need to implement are illustrated in figure 1.

The `onCreate()` method initializes and starts the `HandlerThread` which provides a process thread separate from the UI thread to handle system-driven actions. In the following example, the `onCreate()` method initializes the `CaptioningManager` and prepares to handle the `ACTION_BLOCKED_RATINGS_CHANGED` and `ACTION_PARENTAL_CONTROLS_ENABLED_CHANGED` actions. These actions describe system intents fired when the user changes the parental control settings, and when there is a change on the list of blocked ratings.

```
@Override
public void onCreate() {
    super.onCreate();
    mHandlerThread = new HandlerThread(getClass()
      .getSimpleName());
    mHandlerThread.start();
    mDbHandler = new Handler(mHandlerThread.getLooper());
    mHandler = new Handler();
    mCaptioningManager = (CaptioningManager)
      getSystemService(Context.CAPTIONING_SERVICE);

    setTheme(android.R.style.Theme_Holo_Light_NoActionBar);

    mSessions = new ArrayList<BaseTvInputSessionImpl>();
    IntentFilter intentFilter = new IntentFilter();
    intentFilter.addAction(TvInputManager
      .ACTION_BLOCKED_RATINGS_CHANGED);
    intentFilter.addAction(TvInputManager
```

```
        .ACTION_PARENTAL_CONTROLS_ENABLED_CHANGED);
    registerReceiver(mBroadcastReceiver, intentFilter);
}
```

See Control Content for more information about working with blocked content and providing parental control. See `TvInputManager` for more system-driven actions that you may want to handle in your TV input service.

The `TvInputService` creates a `TvInputService.Session` that implements `Handler.Callback` to handle player state changes. With `onSetSurface()`, the `TvInputService.Session` sets the `Surface` with the video content. See Integrate Player with Surface for more information about working with `Surface` to render video.

The `TvInputService.Session` handles the `onTune()` event when the user selects a channel, and notifies the system TV app for changes in the content and content meta data. These `notify()`code> methods are described in Control Content and Handle Track Selection further in this training.

Define Setup and Settings Activities

The system TV app works with the setup and settings activities you define for your TV input. The setup activity is required and must provide at least one channel record for the system database. The system TV app will invoke the setup activity when it cannot find a channel for the TV input.

The setup activity describes to the system TV app the channels made available through the TV input, as demonstrated in the next lesson, Creating and Updating Channel Data.

The settings activity is optional. You can define a settings activity to turn on parental controls, enable closed captions, set the display attributes, and so forth.

Next Get started Next class

Working with Channel Data

Previous Next Get started

This lesson teaches you to

1. Get Permission
2. Register Channels in the Database
3. Update Channel Data

Try It Out

- TV Input Service sample app

Your TV input must provide Electronic Program Guide (EPG) data for at least one channel in its setup activity. You should also periodically update that data, with consideration for the size of the update and the processing thread that handles it. This lesson discusses creating and updating channel and program data on the system database with these considerations in mind.

Get Permission

In order for your TV input to work with EPG data, it must declare the read and write permissions in its Android manifest file as follows:

```
<uses-permission
android:name="com.android.providers.tv.permission.READ_EPG_DATA" />
<uses-permission
android:name="com.android.providers.tv.permission.WRITE_EPG_DATA" />
```

Register Channels in the Database

The Android TV system database maintains records of channel data for TV inputs. In your setup activity, for each of your channels, you must map your channel data to the following fields of the `TvContract.Channels` class:

- `COLUMN_DISPLAY_NAME` - the displayed name of the channel
- `COLUMN_DISPLAY_NUMBER` - the displayed channel number
- `COLUMN_INPUT_ID` - the ID of the TV input service
- `COLUMN_SERVICE_TYPE` - the channel's service type
- `COLUMN_TYPE` - the channel's broadcast standard type
- `COLUMN_VIDEO_FORMAT` - the default video format for the channel

Although the TV input framework is generic enough to handle both traditional broadcast and over-the-top (OTT) content without any distinction, you may want to define the following columns in addition to those above to better identify traditional broadcast channels:

- `COLUMN_ORIGINAL_NETWORK_ID` - the television network ID
- `COLUMN_SERVICE_ID` - the service ID
- `COLUMN_TRANSPORT_STREAM_ID` - the transport stream ID

For internet streaming based TV inputs, assign your own values to the above accordingly so that each channel can be identified uniquely.

Pull your channel metadata (in XML, JSON, or whatever) from your backend server, and in your setup activity map the values to the system database as follows:

```
ContentValues values = new ContentValues();

values.put(Channels.COLUMN_DISPLAY_NUMBER, channel.mNumber);
values.put(Channels.COLUMN_DISPLAY_NAME, channel.mName);
values.put(Channels.COLUMN_ORIGINAL_NETWORK_ID,
channel.mOriginalNetworkId);
values.put(Channels.COLUMN_TRANSPORT_STREAM_ID,
channel.mTransportStreamId);
values.put(Channels.COLUMN_SERVICE_ID, channel.mServiceId);
values.put(Channels.COLUMN_VIDEO_FORMAT, channel.mVideoFormat);

Uri uri =
context.getContentResolver().insert(TvContract.Channels.CONTENT_URI,
values);
```

channel

In the example above, is an object which holds channel metadata from the backend server.

Present Channel and Program Information

The system TV app presents channel and program information to users as they flip through channels, as shown in figure 1. To make sure the channel and program information works with the system TV app's channel and program information presenter, follow the guidelines below.

1. **Channel number (`COLUMN_DISPLAY_NUMBER`)**
2. **Icon (`android:icon` in the TV input's manifest)**
3. **Program description (`COLUMN_SHORT_DESCRIPTION`)**
4. **Program title (`COLUMN_TITLE`)**
5. **Channel logo (`TvContract.Channels.Logo`)**
    - **Use the color #EEEEEE to match the surrounding text**
    - **Don't include padding**
6. **Poster art (`COLUMN_POSTER_ART_URI`)**
    - **Aspect ratio between 16:9 and 4:3**

Figure 1. The system TV app channel and program information presenter.

The system TV app provides the same information through the program guide, including poster art, as

shown in figure 2.

Figure 2. The system TV app program guide.

Update Channel Data

When updating existing channel data, use the `update()` method instead of deleting and re-adding the data. You can identify the current version of the data by using `Channels.COLUMN_VERSION_NUMBER` and `Programs.COLUMN_VERSION_NUMBER` when choosing the records to update.

Note: Adding channel data to the `ContentProvider` can take time. Only add current programs (those within two hours of the current time) when you update, and use a Sync Adapter to update the rest of the channel data in the background. See the Android TV Live TV Sample App for an example.

Batch Loading Channel Data

When updating the system database with a large amount of channel data, use the `ContentResolver` `applyBatch()` or `bulkInsert()` method. Here's an example using `applyBatch()`:

```
ArrayList<ContentProviderOperation> ops = new ArrayList<>();
int programsCount = mChannelInfo.mPrograms.size();
for (int j = 0; j < programsCount; ++j) {
    ProgramInfo program = mChannelInfo.mPrograms.get(j);
    ops.add(ContentProviderOperation.newInsert(
            TvContract.Programs.CONTENT_URI)
```

```
            .withValues(programs.get(j))
            .withValue(Programs.COLUMN_START_TIME_UTC_MILLIS,
                    programStartSec * 1000)
            .withValue(Programs.COLUMN_END_TIME_UTC_MILLIS,
                    (programStartSec + program.mDurationSec) * 1000)
            .build());
    programStartSec = programStartSec + program.mDurationSec;
    if (j % 100 == 99 || j == programsCount - 1) {
        try {
            getContentResolver().applyBatch(TvContract.AUTHORITY,
ops);
        } catch (RemoteException | OperationApplicationException e)
{
            Log.e(TAG, "Failed to insert programs.", e);
            return;
        }
        ops.clear();
    }
}
```

Processing Channel Data Asynchronously

Data manipulation, such as fetching a stream from the server or accessing the database, should not block the UI thread. Using an AsyncTask is one way to perform updates asynchronously. For example, when loading channel info from a backend server, you can use AsyncTask as follows:

```
private static class LoadTvInputTask extends AsyncTask<Uri, Void,
Void>> {

    private Context mContext;

    public LoadTvInputTask(Context context) {
        mContext = context;
    }

    @Override
    protected Void doInBackground(Uri... uris) {
        try {
            fetchUri(uris[0]);
        } catch (IOException e) {
          Log.d(“LoadTvInputTask”, “fetchUri error”);
        }
        return null;
    }

    private void fetchUri(Uri videoUri) throws IOException {
        InputStream inputStream = null;
        try {
            inputStream =
mContext.getContentResolver().openInputStream(videoUri);
            XmlPullParser parser = Xml.newPullParser();
            try {
```

```
parser.setFeature(XmlPullParser.FEATURE_PROCESS_NAMESPACES, false);
                parser.setInput(inputStream, null);
                sTvInput = ChannelXMLParser.parseTvInput(parser);
                sSampleChannels =
ChannelXMLParser.parseChannelXML(parser);
            } catch (XmlPullParserException e) {
                e.printStackTrace();
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
        }
    }
}
```

If you need to update EPG data on a regular basis, consider using a Sync Adapter or `JobScheduler` to run the update process during idle time, such as every day at 3:00 a.m. See the Android TV live TV sample app for an example.

Other techniques to separate the data update tasks from the UI thread include using the `HandlerThread` class, or you may implement your own using `Looper` and `Handler` classes. See Processes and Threads for more information.

Next Get started Next class

Managing User Interaction

Previous Next Get started

This lesson teaches you to

1. Integrate Player with Surface
2. Use an Overlay
3. Control Content
4. Handle Track Selection

Try It Out

- TV Input Service sample app

In the live TV experience the user changes channels and is presented with channel and program information briefly before the information disappears. Other types of information, such as messages ("DO NOT ATTEMPT AT HOME"), subtitles, or ads may need to persist. As with any TV app, such information should not interfere with the program content playing on the screen.

Figure 1. An overlay message in a live TV app.

Also consider whether certain program content should be presented, given the content's rating and parental control settings, and how your app behaves and informs the user when content is blocked or unavailable. This lesson describes how to develop your TV input's user experience for these considerations.

Integrate Player with Surface

Your TV input must render video onto a `Surface` object, which is passed by the `TvInputService.Session.onSetSurface()` method. Here's an example of how to use a `MediaPlayer` instance for playing content in the `Surface` object:

```
@Override
public boolean onSetSurface(Surface surface) {
    if (mPlayer != null) {
        mPlayer.setSurface(surface);
    }
    mSurface = surface;
    return true;
}

@Override
public void onSetStreamVolume(float volume) {
    if (mPlayer != null) {
        mPlayer.setVolume(volume, volume);
```

```
    }
    mVolume = volume;
}
```

Similarly, here's how to do it using ExoPlayer:

```
@Override
public boolean onSetSurface(Surface surface) {
    if (mPlayer != null) {
        mPlayer.sendMessage(mVideoRenderer,
                MediaCodecVideoTrackRenderer.MSG_SET_SURFACE,
                surface);
    }
    mSurface = surface;
    return true;
}

@Override
public void onSetStreamVolume(float volume) {
    if (mPlayer != null) {
        mPlayer.sendMessage(mAudioRenderer,
                MediaCodecAudioTrackRenderer.MSG_SET_VOLUME,
                volume);
    }
    mVolume = volume;
}
```

Use an Overlay

Use an overlay to display subtitles, messages, ads or MHEG-5 data broadcasts. By default, the overlay is disabled. You can enable it when you create the session by calling `TvInputService.Session.setOverlayViewEnabled(true)`, as in the following example:

```
@Override
public final Session onCreateSession(String inputId) {
    BaseTvInputSessionImpl session =
onCreateSessionInternal(inputId);
    session.setOverlayViewEnabled(true);
    mSessions.add(session);
    return session;
}
```

Use a `View` object for the overlay, returned from `TvInputService.Session.onCreateOverlayView()`, as shown here:

```
@Override
public View onCreateOverlayView() {
    LayoutInflater inflater = (LayoutInflater)
getSystemService(LAYOUT_INFLATER_SERVICE);
    View view = inflater.inflate(R.layout.overlayview, null);
```

```
    mSubtitleView = (SubtitleView) view.findViewById(R.id.subtitles);

    // Configure the subtitle view.
    CaptionStyleCompat captionStyle;
    float captionTextSize = getCaptionFontSize();
    captionStyle = CaptionStyleCompat.createFromCaptionStyle(
            mCaptioningManager.getUserStyle());
    captionTextSize *= mCaptioningManager.getFontScale();
    mSubtitleView.setStyle(captionStyle);
    mSubtitleView.setTextSize(captionTextSize);
    return view;
}
```

The layout definition for the overlay might look something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"

    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <com.google.android.exoplayer.text.SubtitleView
        android:id="@+id/subtitles"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom|center_horizontal"
        android:layout_marginLeft="16dp"
        android:layout_marginRight="16dp"
        android:layout_marginBottom="32dp"
        android:visibility="invisible"/>
</FrameLayout>
```

Control Content

When the user selects a channel, your TV input handles the `onTune()` callback in the `TvInputService.Session` object. The system TV app's parental controls determine what content displays, given the content rating. The following sections describe how to manage channel and program selection using the `TvInputService.Session` `notify` methods that communicate with the system TV app.

Make Video Unavailable

When the user changes the channel, you want to make sure the screen doesn't display any stray video artifacts before your TV input renders the content. When you call `TvInputService.Session.onTune()`, you can prevent the video from being presented by calling

TvInputService.Session.notifyVideoUnavailable() and passing the VIDEO_UNAVAILABLE_REASON_TUNING constant, as shown in the following example.

```
@Override
public boolean onTune(Uri channelUri) {
    if (mSubtitleView != null) {
        mSubtitleView.setVisibility(View.INVISIBLE);
    }

notifyVideoUnavailable(TvInputManager.VIDEO_UNAVAILABLE_REASON_TUNING);

    mUnblockedRatingSet.clear();

    mDbHandler.removeCallbacks(mPlayCurrentProgramRunnable);
    mPlayCurrentProgramRunnable = new
PlayCurrentProgramRunnable(channelUri);
    mDbHandler.post(mPlayCurrentProgramRunnable);
    return true;
}
```

Then, when the content is rendered to the Surface, you call TvInputService.Session.notifyVideoAvailable() to allow the video to display, like so:

```
@Override
public void onDrawnToSurface(Surface surface) {
    mFirstFrameDrawn = true;
    notifyVideoAvailable();
}
```

This transition lasts only for fractions of a second, but presenting a blank screen is visually better than allowing the picture to flash odd blips and jitters.

See also, Integrate Player with Surface for more information about working with Surface to render video.

Provide Parental Control

To determine if a given content is blocked by parental controls and content rating, you check the TvInputManager class methods, isParentalControlsEnabled() and isRatingBlocked(android.media.tv.TvContentRating). You might also want to make sure the content's TvContentRating is included in a set of currently allowed content ratings. These considerations are shown in the

following sample.

```
private void checkContentBlockNeeded() {
    if (mCurrentContentRating == null ||
!mTvInputManager.isParentalControlsEnabled()
            ||
!mTvInputManager.isRatingBlocked(mCurrentContentRating)
            || mUnblockedRatingSet.contains(mCurrentContentRating)) {
        // Content rating is changed so we don't need to block
anymore.
        // Unblock content here explicitly to resume playback.
        unblockContent(null);
        return;
    }

    mLastBlockedRating = mCurrentContentRating;
    if (mPlayer != null) {
        // Children restricted content might be blocked by TV app as
well,
        // but TIF should do its best not to show any single frame
of blocked content.
        releasePlayer();
    }

    notifyContentBlocked(mCurrentContentRating);
}
```

Once you have determined if the content should or should not be blocked, notify the system TV app by calling the `TvInputService.Session` method `notifyContentAllowed()` or `notifyContentBlocked()` , as shown in the previous example.

Use the `TvContentRating` class to generate the system-defined string for the `COLUMN_CONTENT_RATING` with the `TvContentRating.createRating()` method, as shown here:

```
TvContentRating rating = TvContentRating.createRating(
    "com.android.tv",
    "US_TV",
    "US_TV_PG",
    "US_TV_D", "US_TV_L");
```

Handle Track Selection

The `TvTrackInfo` class holds information about media tracks such as the track type (video, audio, or subtitle) and so forth.

The first time your TV input session is able to get track information, it should call `TvInputService.Session.notifyTracksChanged()` with a list of

all tracks to update the system TV app. When there is a change in track information, call `notifyTracksChanged()` again to update the system.

The system TV app provides an interface for the user to select a specific track if more than one track is available for a given track type; for example, subtitles in different languages. Your TV input responds to the `onSelectTrack()` call from the system TV app by calling `notifyTrackSelected()` , as shown in the following example. Note that when `null` is passed as the track ID, this *deselects* the track.

```
@Override
public boolean onSelectTrack(int type, String trackId) {
    if (mPlayer != null) {
        if (type == TvTrackInfo.TYPE_SUBTITLE) {
            if (!mCaptionEnabled && trackId != null) {
                return false;
            }
            mSelectedSubtitleTrackId = trackId;
            if (trackId == null) {
                mSubtitleView.setVisibility(View.INVISIBLE);
            }
        }
        if (mPlayer.selectTrack(type, trackId)) {
            notifyTrackSelected(type, trackId);
            return true;
        }
    }
    return false;
}
```

Next Get started Next class

TV Apps Checklist

Checklist sections

- TV Form Factor Support
- User Interface Design
- Search and Content Discovery
- Games

You should also read

- TV App Quality
- Material Design on Android Checklist
- Launch Checklist

Users enjoy the TV app experience when it is consistent, logical, and predictable. They should be able to navigate within your app and throughout Android TV without getting lost or having to "reset" the UI and start over. Users appreciate clear, colorful, and functional interfaces that make the experience magical. With these ideas in mind, you can create an app that fits nicely in Android TV and performs as users expect.

This checklist covers the main aspects of development for both apps and games and provides guidelines to ensure that your app provides the best possible experience. Additional considerations for games only are covered in the *Games* section.

For criteria that qualify an Android TV app on Google Play, see TV App Quality.

TV Form Factor Support

These checklist items apply to **Games** and **Apps**.

1. Identify the main TV activity with the `CATEGORY_LEANBACK_LAUNCHER` filter in the manifest.

   See Declare a TV Activity.

2. Provide a home screen banner for each language supported by your app
   - Launcher app banner measures 320x180 px
   - Banner resource is located in the `drawables/xhdpi` directory
   - Banner image includes localized text to identify the app.

   See Provide a home screen banner.

3. Eliminate requirements for unsupported hardware in your app.

   See Declaring hardware requirements for TV.

4. Ensure permissions do not imply hardware requirements

   See Declaring permissions that imply hardware features.

User Interface Design

These checklist items apply to **Games** and **Apps**.

1. Provide appropriate layout resources for landscape mode.

   See Build Basic TV Layouts.

2. Ensure that text and controls are large enough to be visible from a distance.

   See Build Useable Text and Controls.

3. Provide high-resolution bitmaps and icons for HDTV screens.

   See Manage Layout Resources for TV.

4. Make sure your icons and logo conform to Android TV specifications.

See Manage Layout Resources for TV.

5. Allow for overscan in your layout.

   See Overscan.

6. Make every UI element work with both D-pad and game controllers.

   See Creating Navigation and Handling Controllers.

7. Change the background image as users browse through content.

   See Update the Background.

8. Customize the background color to match your branding in Leanback fragments.

   See Customize the Card View.

9. Ensure that your UI does not require a touch screen.

   See Touch screen and Declare touch screen not required.

10. Follow guidelines for effective advertising.

    See Provide Effective Advertising.

Search and Content Discovery

These checklist items apply to **Games** and **Apps**.

1. Provide search results from your app in the Android TV global search box.

   See Provide Data.

2. Provide TV-specific data fields for search.

   See Identify Columns.

3. Make sure your app presents discovered content in a details screen that lets the user start watching the content immediately.

   See Display Your App in the Details Screen.

4. Put relevant, actionable content and categories on the main screen, making it easy to discover content.

   See Recommending TV Content.

Games

These checklist items apply to **Games**.

1. Show your game on the home screen with the `isGame` flag in the manifest.

   See Show your game on the home screen.

2. Make sure game controller support does not depend upon the Start, Select, or Menu buttons (not all controllers have these).

   See Input Devices.

3. Use a generic gamepad graphic (without specific controller branding) to show game button mappings.

   See Show controller instructions.

4. Check for both ethernet and WiFi connectivity.

   See Networking.

5. Provide users with a clean exit.

   See Exit.

Next Get started Next class

Building Apps for Auto

The Android platform enables you to extend your app to work with in-vehicle console systems running Android Auto. These systems provide a simplified interface for apps that can be used in a car, allowing users to take your app with them on the way to the grocery store or on a long road trip.

Apps that work with Android Auto consoles run on a connected device, such as a phone or tablet. The app communicates via specific APIs with the in-dash console, which provides a user interface for the connected app that is designed for use in a car.

For more information, follow the links below to learn how to extend your Android app to support use in vehicles.

Get Started

Learn the basics of extending your app for use in vehicles, with information about app configuration and user interface design considerations:

Audio Apps

Learn how to extend your audio app to enable content navigation and playback through a vehicle console:

Messaging Apps

Learn how to extend your messaging app to provide text communication services through a vehicle console:

App Quality and Distribution

Learn how to distribute your Auto app and how apps are evaluated for quality and driver safety:

Next Get started Next class

Getting Started with Auto

Dependencies and Prerequisites

- Android 5.0 (API level 21) or higher

This class teaches you how to

1. Set Up an Auto Project
2. Build Auto Apps
3. Run and Test Auto Apps

You should also read

- Designing for Auto
- Providing Audio Playback with Auto
- Providing Messaging for Auto

Android Auto extends the Android platform into the car. When users connect their handheld devices running Android 5.0 or higher to a compatible vehicle, the Auto user interface provides a car-optimized Android experience on the vehicle's screen. Users interact with compatible apps and services through voice actions and the vehicle's input controls (like a touchscreen or dashboard buttons).

Auto currently supports two types of apps:

- *Audio apps* that allow users to browse and play music and spoken audio content in the car.
- *Messaging apps* that receive incoming notifications, read messages aloud via text-to-speech, and send replies via voice input in the car.

You can enable your existing audio and messaging apps developed for

phones and tablets to work in the car, without having to worry about vehicle-specific hardware differences. To enable your app for Auto, your app must target Android 5.0 (API level 21) or higher. Your app's manifest must also declare the car capabilities that it uses, such as audio playback or messaging services.

This lesson describes how to start building apps for Auto, including setting up your development environment and meeting the the minimum requirements to enable an app to communicate with Auto.

Set Up an Auto Project

This section describes how to create a new app or modify an existing app to communicate with Auto.

Prerequisites

Before you begin building apps for Auto, you must:

- **Create or update your app project** - Android 5.0 (API level 21) provides new APIs for implementing audio playback and messaging that is compatible with Auto. To access the new APIs, create a project or modify an existing project to target Android 5.0 (API level 21) or higher. This means you must set the manifest `targetSdkVersion` to 21 or higher.
- **Install the support library** - If you are building messaging apps for Auto, you need the `NotificationCompat.CarExtender` class contained in the v4 support library. This class allows you to create notifications that are compatible with Auto devices.

Declare Auto capabilities

The Auto features that your app can access are controlled by the settings in your app manifest and a separate XML configuration file. Before adding Auto features to your app, you must first define the Auto XML configuration file and add a manifest entry referencing your XML file.

Define the Auto XML configuration file

Specify the car capabilities that your app uses in an XML file that you place in your project's resources directory (`res/xml/`). For example, to extend an audio application for Auto, create a file called `automotive_app_desc.xml` and store it under your projects's `res/xml/` folder. The `automotive_app_desc.xml` file contains the following metadata:

```
<automotiveApp>
   <uses name="media" />
</automotiveApp>
```

The `<uses>` element declares the Auto capability your app intends to use. Multiple `<uses>` tags can be added if your application uses multiple car capabilities. The `name` attribute indicates the specific capability your app uses. The values supported are:

- `media` - The app uses the Android framework APIs to play music in a vehicle. Set this value if you are enabling an audio app for Auto.
- `notification` - The app displays message notifications in the car's Overview screen, allows users select a message to be read aloud, and lets them respond through voice input. Set this value if you are enabling a messaging app for Auto.

Add a manifest entry

In your app's manifest (`AndroidManifest.xml`), provide a reference to the Auto XML configuration file you created in the previous section. Add a `"com.google.android.gms.car.application"` metadata entry under the `<application>` element that references your Auto XML configuration file. Omit the `.xml` file extension when specifying the configuration filename.

The following code snippet shows how to include this reference in your manifest.

```
<application>

    ...
    <meta-data android:name="com.google.android.gms.car.application"
     android:resource="@xml/automotive_app_desc"/>

</application>
```

Add Auto Features to Your Apps

After you have completed the steps described above, you're ready to add Auto features to your apps. See these additional topics to help you build apps for Auto:

- Providing Audio Playback for Auto - Create apps that let users browse and play music in the car.
- Providing Messaging for Auto - Enable users to receive and reply to messages in the car.

Important: Google takes driver distraction very seriously. There are specific design requirements your app must meet to qualify as an Auto app on Google Play. By adhering to these requirements, you can reduce the effort for building and testing your app. For more information, see Auto App Quality.

Run and Test Auto Apps

As you prepare to publish your app, make sure that your app looks correct when projected on the Auto user interface. Use the Android Media Browser simulator and Android Messaging simulators to view and test your audio or messaging apps in a screen that looks similar to what is projected on Auto.

To get the simulators, open the SDK Manager and download them from **Extras > Android Auto API Simulators**.

Before you begin testing, compile your app in your development environment. Install your app and the Android simulator for the features you want to test (that is, audio or messaging) on a physical or virtual device running Android 5.0 (API level 21) or higher. To check the version of Android on the device, go to **Settings > About > Android Version**.

Testing audio apps

To run and test audio apps:

1. Install the Android Media Browser simulator (`<sdk>/extras/google/simulators/media-browser-simulator.apk`) on the test device. You can do this using the adb command line tool.
2. Enable developer options on the test device.
3. Install your app on the test device.
4. Launch the Android Media Browser simulator to see how your audio app appears in Auto. If your app does not appear, stop the simulator from **Settings > Apps** then restart it.

Testing messaging apps

To run and test messaging apps:

1. Install the Android Messaging simulator (`<sdk>/extras/google/simulators/messaging-simulator.apk`) on the test device. You can do this using the adb command line tool.
2. Enable the simulator to read notifications posted on the system:
   a. Enable developer options on the test device.
   b. Click **Settings > Sounds & Notifications > Notification Access** and check the box labeled **Messaging Simulator**.
3. Install your app on the test device.
4. Launch the Android Messaging Simulator to see how your messaging app appears in Auto. If your app does not appear, stop the simulator from **Settings > Apps** then restart it.

Next Get started Next class

Providing Audio Playback for Auto

Dependencies and Prerequisites

- Android 5.0 (API level 21) or higher

This class teaches you how to

1. Provide Audio Services
2. Configure Your Manifest
3. Build a Browser Service
4. Implement Play Controls
5. Support Voice Actions

Related Samples

- MediaBrowserService
- Universal Media Player

See Also

- User Experience Guidelines: Audio Apps
- Managing Audio Playback
- ExoPlayer

Video

Devbytes: Android Auto Audio

Drivers want to access their music and other audio content on the road.

Audio books, podcasts, sports commentary, and recorded talks can make a long trip educational, inspirational, and enjoyable. The Android framework allows you to extend your audio app so users can listen to their favorite tunes and audio content using a simple, yet customizable user interface.

Apps running on mobile devices with Android 5.0 or higher can provide audio services for dashboard systems running Android Auto. By configuring your app with a few settings and implementing a service for accessing music tracks, you can enable Auto devices to discover your app and provide a browse and playback interface for your app's audio content.

This class assumes that you have built an app that plays audio through an Android device's integrated speakers or connected headphones. It describes how to extend your app to allow Auto devices to browse your content listings and play it through a car stereo system.

Provide Audio Services

Audio apps do not directly control a car dashboard device that runs Android Auto. When the user connects an Android mobile device into a dashboard system, Android Auto discovers your app through manifest entries that indicate what audio services your app can provide. The dashboard system displays a launcher icon for your app as a music provider and the user can choose to use your app's services. If the user launches your app, the Auto device queries your app to see what content is available, displays your content items to the user, and sends requests to your app to control playback with actions such as play, pause, or skip track.

To enable your app to provide audio content for Auto devices, you need to:

- Configure your app manifest to do the following:
    - Declare that your app can provide audio content for Auto devices.

Define a service that provides a browsable list of your audio tracks.

- Build a service that provides audio track listing information extending `MediaBrowserService`.
- Register a `MediaSession` object and implement the `MediaSession.Callback` object to enable playback controls.

Configure Your Manifest

When a user plugs an Android mobile device into a dashboard device running Auto, the system requests a list of installed apps that include app manifest entries to indicate they support services for Auto devices and how to access them. This section describes how to configure your app manifest to indicate your app supports audio services for Auto devices, and allow dashboard system to connect with your app.

Declare Auto audio support

You indicate that your app supports cars capabilities using the following manifest entry:

```
<application>
    ...
    <meta-data android:name="com.google.android.gms.car.application"
        android:resource="@xml/automotive_app_desc"/>
    ...
<application>
```

This manifest entry refers to a secondary XML file, where you declare what Auto capabilities your app supports. For an app that supports audio for cars, add an XML file to the `res/xml/` resources directory as `automotive_app_desc.xml`, with the following content:

```
<automotiveApp>
    <uses name="media"/>
</automotiveApp>
```

For more information about declaring capabilities for Auto devices, see Getting Started with Auto.

Declare your media browser service

Auto devices expect to connect to a service in order to browse audio track listings. You declare this service in your manifest to allow the dashboard system to discover this service and connect to your app.

The following code example shows how to declare this listing browser service in your manifest:

```
<application>
    ...
    <service android:name=".MyMediaBrowserService"
                android:exported="true">
        <intent-filter>
            <action android:name=
                "android.media.browse.MediaBrowserService"/>
        </intent-filter>
    </service>
    ...
<application>
```

The service your app provides for browsing audio tracks must extend the `MediaBrowserService`. The implementation of this service is discussed in the Build a Browser Service section.

Note: Other clients can also contact your app's browser service aside from Auto devices. These media clients might be other apps on a user's mobile device, or they might be other remote clients.

Specify a notification icon

The Auto user interface shows notifications about your audio app to the user during the course of operation. For example, if the user has a navigation app running, and one song finishes and a new song starts, the Auto device shows the user a notification to indicate the change with an icon from your app. You can specify an icon that is used to represent your app for these notifications using the following manifest declaration:

```
<application>
    ...
    <meta-data
android:name="com.google.android.gms.car.notification.SmallIcon"
        android:resource="@drawable/ic_notification" />
    ...
<application>
```

Note: The icon you provide should have transparency enabled, so the

icon's background gets filled in with the app's primary color.

Build a Browser Service

Auto devices interact with your app by contacting its implementation of a `MediaBrowserService`, which you declare in your app manifest. This service allows Auto devices to find out what content your app provides. Connected Auto devices can also query your app's media browser service to contact the `MediaSession` provided by your app, which handles content playback commands.

You create a media browser service by extending the `MediaBrowserService` class. Connected Auto devices can contact your service to do the following:

- Browse your app's content hierarchy, in order to present a menu to the user
- Get the token for your app's `MediaSession` object, in order to control audio playback

Media browser service workflow

1. When your app's audio services are requested by a user through a connected Auto device, the dashboard system contacts your app's media browser service. In your implementation of the `onCreate()` method, you must create and register a `MediaSession` object and its callback object.
2. The Auto device calls the browser service's `onGetRoot()` method to get the top node of your content hierarchy. The node retrieved by this call is not used as a menu item, it is only used to retrieve its child nodes, which are subsequently displayed as the top menu items.
3. Auto invokes the `onLoadChildren()` method to get the children of the root node, and uses this information to present a menu to the user.
4. If the user selects a submenu, Auto invokes `onLoadChildren()` again to retrieve the child nodes of the selected menu item.
5. If the user begins playback, Auto invokes the appropriate media

session callback method to perform that action. For more information, see the section about how to Implement Playback Controls.

Building your content hierarchy

Auto devices acting as audio clients call your app's `MediaBrowserService` to find out what content you have available. You need to implement two methods in your browser service to support this: `onGetRoot()` and `onLoadChildren()`.

Each node in your content hierarchy is represented by a `MediaBrowser.MediaItem` object. Each of these objects is identified by a unique ID string. The client treats these ID strings as opaque tokens. When a client wants to browse to a submenu, or play a content item, it passes the ID token. Your app is responsible for associating the ID token with the appropriate menu node or content item.

Note: You should consider providing different content hierarchies depending on what client is making the query. In particular, Auto applications have strict limits on how large a menu they can display. This is intended to minimize distracting the driver, and to make it easy for the driver to operate the app via voice commands. For more information on the Auto user experience restrictions, see the Auto Audio Apps guidelines.

Your implementation of `onGetRoot()` returns information about the root node of the menu hierarchy. This root node is the parent of the top items your browse hierarchy. The method is passed information about the calling client. You can use this information to decide if the client should have access to your content at all. For example, if you want to limit your app's content to a list of approved clients, you can compare the passed `clientPackageName` to your whitelist. If the caller isn't an approved package, you can return null to deny access to your content.

A typical implementation of `onGetRoot()` might look like this:

```
@Override
public BrowserRoot onGetRoot(String clientPackageName, int
clientUid,
```

```
    Bundle rootHints) {

    // To ensure you are not allowing any arbitrary app to browse
your app's
    // contents, you need to check the origin:
    if (!PackageValidator.isCallerAllowed(this, clientPackageName,
clientUid)) {
        // If the request comes from an untrusted package, return
null.
        // No further calls will be made to other media browsing
methods.
        LogHelper.w(TAG, "OnGetRoot: IGNORING request from untrusted
package "
                + clientPackageName);
        return null;
    }
    if (ANDROID_AUTO_PACKAGE_NAME.equals(clientPackageName)) {
        // Optional: if your app needs to adapt ads, music library
or anything
        // else that needs to run differently when connected to the
car, this
        // is where you should handle it.
    }
    return new BrowserRoot(MEDIA_ID_ROOT, null);
}
```

The Auto device client builds the top-level menu by calling `onLoadChildren()` with the root node object and getting it's children. The client builds submenus by calling the same method with other child nodes. The following example code shows a simple implementation of `onLoadChildren()` method:

```
@Override
public void onLoadChildren(final String parentMediaId,
    final Result<List<MediaItem>> result) {

    // Assume for example that the music catalog is already
loaded/cached.

    List<MediaBrowser.MediaItem> mediaItems = new ArrayList<>();

    // Check if this is the root menu:
    if (MEDIA_BROWSER_ROOT.equals(parentMediaId)) {

        // build the MediaItem objects for the top level,
        // and put them in the <result> list
    } else {

        // examine the passed parentMediaId to see which submenu
we're at,
        // and put the children of that menu in the <result> list
    }
}
```

Enable Playback Control

Auto devices use `MediaSession` objects to pass playback control commands to an app that is providing audio services. Your audio app must create an instance of this object to pass to the dashboard device and implement callback methods to enable remote control of audio playback.

Register a media session

An Auto device using your app as audio service needs to obtain a `MediaSession` object from your app. The Auto device uses the session object to send playback commands requested by the Auto user back to your app.

When you initialize your browser service, you register that session object with your `MediaBrowserService` by calling the `setSessionToken()` method. This step allows clients such as an Auto device to retrieve that object by calling your browser service's `getSessionToken()` method.

In your browser service's `onCreate()` method, create a `MediaSession`. You can then query the `MediaSession` to get its token, and register the token with your browser service:

```
public void onCreate() {
    super.onCreate();

    ...
    // Start a new MediaSession
    MediaSession mSession = new MediaSession(this, "session tag");
    setSessionToken(mSession.getSessionToken());

    // Set a callback object to handle play control requests, which
    // implements MediaSession.Callback
    mSession.setCallback(new MyMediaSessionCallback());

    ...
```

When you create the media session object, you set a callback object that is used to handle playback control requests. You create this callback object by providing an implementation of the `MediaSession.Callback` class for your app. The next section discusses how to implement this object.

Implement play commands

When an Auto device requests playback of an audio track from your app, it uses the `MediaSession.Callback` class from your app's `MediaSession` object, which it obtained from your app's media browse service. When an Auto user wants to play content or control content playback, such as pausing play or skipping to the next track, Auto invokes one of the callback object's methods.

To handle content playback, your app must extend the abstract `MediaSession.Callback` class and implement the methods that your app supports. The most important callback methods are as follows:

`onPlay()`
: Invoked if the user chooses play without choosing a specific item. Your app should play its default content. If playback was paused with `onPause()`, your app should resume playback.

Note: Google Play requires your app not to play music immediately when it launches. For more information on this and other requirements, see Auto App Quality.

`onPlayFromMediaId()`
: Invoked when the user chooses to play a specific item. The method is passed the item's media ID, which you assigned to the item in the content hierarchy.

`onPlayFromSearch()`
: Invoked when the user chooses to play from a search query. The app should make an appropriate choice based on the passed search string.

`onPause()`
: Pause playback.

`onSkipToNext()`
: Skip to the next item.

`onSkipToPrevious()`
: Skip to the previous item.

`onStop()`
: Stop playback.

Your app should override these methods to provide any desired

functionality. In some cases you might not implement a method if it is not supported by your app. For example, if your app plays a live stream (such as a sports broadcast), the skip to next function might not make sense. In that case, you could simply use the default implementation of `onSkipToNext()`.

When your app receives a request to play content, it should play audio the same way it would in a non-Auto situation (as if the user was listening through a device speaker or connected headphones). The audio content is automatically sent to the dashboard system to be played over the car's speakers.

For more information about playing audio content, see Media Playback, Managing Audio Playback, and ExoPlayer.

Support Voice Actions

To reduce driver distractions, you can add voice actions in your audio playback app. With voice action support, users can launch your app and play audio by providing voice input on Auto screens. If your audio playback app is already active and the user says *"Play a song"*, the system starts playing music without requiring the user to look at or touch the screen.

Enable your app to handle audio playback requests

Enable your audio app to launch with a voice command such as *"Play [search query] on [your app name]"* by adding the following entry in your manifest:

```
<activity>
    <intent-filter>
        <action android:name=
             "android.media.action.MEDIA_PLAY_FROM_SEARCH" />
        <category android:name=
             "android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
```

When the user says *"Play music on [your app name]"* on an Auto screen,

Auto attempts to launch your app and play audio by calling your app's `MediaSession.Callback.onPlayFromSearch()` method. If the user has not specified criteria such as a track name or music genre, the `MediaSession.Callback.onPlayFromSearch()` method receives an empty query parameter. Your app should respond by immediately playing audio, such as a song from a random queue or the most recent playlist.

Parse the voice query to build the playback queue

When a user searches for a specific criteria, such as *"Play jazz on [your app name]"* or *"Listen to [song title]"*, the `onPlayFromSearch()` callback method receives the voice search results in the query parameter and an extras bundle. For more information on how to handle search queries to play audio content, see Play music based on a search query.

To parse the voice search query to play back audio content in your app, follow these steps:

1. Use the extras bundle and search query string returned from the voice search to filter results.
2. Build the audio content queue based on these results.
3. Play the audio content.

The `onPlayFromSearch()` method takes an extras parameter with more detailed information from the voice search. These extras help you find the audio content in your app for playback. If the search results are unable to provide this data, you can implement logic to parse the raw search query and play the appropriate tracks based on the query.

The following extras are supported in Android Auto:

- `android.intent.extra.album`
- `android.intent.extra.artist`
- `android.intent.extra.genre`
- `android.intent.extra.playlist`
- `android.intent.extra.title`

The following snippet shows how to override the `onPlayFromSearch()`

method in your `MediaSession.Callback` implementation to handle the search query and extras for playing audio content in your app:

```
@Override
public void onPlayFromSearch(String query, Bundle extras) {
    if (TextUtils.isEmpty(query)) {
        // The user provided generic string e.g. 'Play music'
        // Build appropriate playlist queue
    } else {
        // Build a queue based on songs that match "query" or
"extras" param
        String mediaFocus =
extras.getString(MediaStore.EXTRA_MEDIA_FOCUS);
        if (TextUtils.equals(mediaFocus,
                MediaStore.Audio.Artists.ENTRY_CONTENT_TYPE)) {
            isArtistFocus = true;
            artist = extras.getString(MediaStore.EXTRA_MEDIA_ARTIST);
        } else if (TextUtils.equals(mediaFocus,
                MediaStore.Audio.Albums.ENTRY_CONTENT_TYPE)) {
            isAlbumFocus = true;
            album = extras.getString(MediaStore.EXTRA_MEDIA_ALBUM);
        }

        // Implement additional "extras" param filtering
    }

    // Implement your logic to retrieve the queue
    if (isArtistFocus) {
        result = searchMusicByArtist(artist);
    } else if (isAlbumFocus) {
        result = searchMusicByAlbum(album);
    }

    if (result == null) {
        // No focus found, search by query for song title
        result = searchMusicBySongTitle(query);
    }

    if (result != null && !result.isEmpty()) {
        // Immediately start playing from the beginning of the
search results
        // Implement your logic to start playing music
        playMusic(result);
    } else {
        // Handle no queue found. Stop playing if the app
        // is currently playing a song
    }
}
```

Note: To minimize driver distractions, immediately initiate audio content playback in the `onPlayFromSearch()` method when you have generated the audio content queue based on the user's request.

For a more detailed example on how to implement voice search to play audio content in your app, see the Universal Media Player sample.

Implement playback control actions

To provide a hands-free experience while users drive and listen to audio content in Android Auto, your app should allow users to control audio content playback with voice actions. When users speak commands such as *“Next song”*, *“Pause music”*, or *“Resume music”*, the system triggers the corresponding callback method where you implement the playback control action.

To provide voice-enabled playback controls, first enable the hardware controls by setting these flags in your app’s `MediaSession` object:

```
mSession.setFlags(MediaSession.FLAG_HANDLES_MEDIA_BUTTONS |
    MediaSession.FLAG_HANDLES_TRANSPORT_CONTROLS);
```

Then, implement the callback methods with the playback controls that you support in your app. Here’s a list of voice-enabled playback controls supported by Android Auto:

Example phrase	Callback method
"Next song"	`onSkipToNext()`
"Previous song"	`onSkipToPrevious()`
"Pause music"	`onPause()`
"Stop music"	`onStop()`
"Resume music"	`onPlay()`

For a more detailed example on how to implement voice-enabled playback actions in your app, see the Universal Media Player sample.

Next Get started Next class

Providing Messaging for Auto

Dependencies and Prerequisites

- Android 5.0 (API level 21) or higher

This class teaches you to:

- Provide Messaging Services
- Configure Your Manifest
- Import Support Library for Messaging
- Notify Users of Messages
- Handle User Actions

Related Samples

- MessagingService

See Also

- User Experience Guidelines: Messaging Apps
- Notifications

Video

DevBytes: Android Auto Messaging

Staying connected through messages is important to many drivers. Chat apps can let users know if a child need to be picked up, or if a dinner location has been changed. The Android framework enables messaging apps to extend their services into car dashboards using a standard user interface that lets drivers keep their eyes on the road.

Apps that support messaging can be extended to pass messaging notifications to Auto dashboard systems, alerting them to new messages and allowing them to respond. You can configure your messaging app to provide these services when an Android mobile device with your app installed is connected to an Auto dashboard. Once connected, your app can provide text information to users and allow them to respond. The Auto dashboard system handles displaying the notification and the interface for replies.

This lesson assumes that you have built an app that displays messages to the user and receive the user's replies, such as a chat app. It shows you how to extend your app to hand those messages off to an Auto device for display and replies.

Provide Messaging Services

Messaging apps do not run directly on the Android dashboard hardware. They are installed on a separate Android mobile device. When the mobile device is plugged into a dashboard, the installed messaging apps can offer services for viewing and responding to messages through the Auto user interface.

To enable your app to provide messaging services for Auto devices:

- Configure your app manifest to indicate that your app provides messaging services which are compatible with Android Auto dashboard devices.
- Build and send a specific type of notification for display on Auto devices.
- Configure your app to receive `Intent` objects that indicate a user has read or replied to a message.

Concepts and objects

Before you start designing your app, it's helpful to understand how Auto handles messaging.

Each individual chunk of communication is a *message*. A message is a short length of text, suitable for the Auto device to read aloud. In a chat app, this might be a single message from one person to another: `"Fitzy -- Jane can't come to the ball, her youngest has the croup. :-( -- Liz"`.

A *conversation* is a group of messages that are all grouped together in some way. Auto uses the conversation information to group the messages together when presenting them to the user. In a chat app, a conversation might be all the messages between the user and another person (for example, all the messages back and forth between Darcy and Elizabeth). Every message belongs to a conversation, even if it's the only message in that conversation. Each conversation has a *conversation name*. The conversation name is used by Android Auto to present the messages; it's up to your app to choose an appropriate conversation name. In a chat app, the conversation name is usually the person your user is talking to.

The v4 support library defines an `UnreadConversation` object. This object holds all messages in a conversation which have not yet been heard by the user. To give those messages to the user, you attach that `UnreadConversation` to a notification. However, you do not attach messages to the `UnreadConversation` directly. Instead, you must first set up an `UnreadConversation.Builder` object for the conversation. The messages are added to the builder, then when you are ready to send the messages, you use the builder to create the actual `UnreadConversation` and attach the `UnreadConversation` to the notification.

Note: When Auto presents messages to the user, it uses the notification *tag* and *ID* to determine which conversation the messages belong to. It is important to use the same tag and ID for all messages in a conversation, and to not use that tag for other conversations.

Workflow

This section describes how the mobile device interacts with Auto to present messages to the user.

1. The app receives a message that it wants to pass on to the user. It attaches the message to an `UnreadConversation` object and attaches it to a notification. That notification is associated with a `CarExtender` object, which indicates that the notification can be handled by Android Auto.
2. The app posts the notification. The Android notification framework passes the message to Auto. Auto uses the notification tag and ID to determine which conversation the message belongs to, and presents the message to the user in an appropriate way.
3. When the user listens to the message, Auto triggers the app's message heard pending intent. The app should discard the `UnreadConversation` object and its builder at this time, since the messages contained in those objects have been heard by the user.
4. If the user sends a reply, Auto triggers the app's "message reply" intent and attaches a transcript of the user's response. The app can take appropriate action, based on the app's logic. For example, a chat app might interpret the reply as a message to go to the other conversation participants.

Configure Your Manifest

You configure your app manifest to indicate that it supports messaging services for Auto devices and handle message actions. This section describes what changes to make to your manifest to support messaging for Auto devices.

Declare Auto messaging support

When a user connects a Android mobile device to a dashboard running Android, the dashboard device looks for apps that declare support for vehicle services, such as messaging. You indicate that your app supports cars capabilities using the following manifest entry:

```
<application>
    ...
    <meta-data android:name="com.google.android.gms.car.application"
        android:resource="@xml/automotive_app_desc" />
    ...
<application>
```

This manifest entry refers to a secondary xml file, where you declare what Auto capabilities your app supports. For an app that supports messaging for Auto devices, add an xml file to the `res/xml/` your app's development project directory as `automotive_app_desc.xml`, with the following content:

```
<automotiveApp>
    <uses name="notification"/>
</automotiveApp>
```

For more information about declaring capabilities for Auto devices, see Getting Started with Auto.

Define read and reply intent filters

Auto devices use `Intent` objects that indicate a user has read or replied to a message provided by your app. Your app defines intent types for reading and replying to messages and adds this information to messaging notifications for Auto devices, so that the dashboard system can notify your app when a user takes one of these actions.

You define the read action and reply action intents types for your app and the `BroadcastReceiver` classes that handle them in the manifest. The following code example demonstrates how to declare these intents and their associated receivers.

```
<application>
    ...
    <receiver android:name=".MyMessageHeardReceiver">
        <intent-filter>
          <action
android:name="com.myapp.messagingservice.MY_ACTION_MESSAGE_HEARD"/>
        </intent-filter>
    </receiver>

    <receiver android:name=".MyMessageReplyReceiver">
        <intent-filter>
          <action
android:name="com.myapp.messagingservice.MY_ACTION_MESSAGE_REPLY"/>
        </intent-filter>
    </receiver>
    ...
</application>
```

In this example, `"MyMessageReadReceiver"` and `"MyMessageReplyReceiver"` are the names of the `BroadcastReceiver` subclasses you define to handle

the intents. You can choose whatever you like as the action names, but it's best to prepend your package name to ensure that the action names are unique. For more information about handling actions, see Handle User Actions.

Import Support Library for Messaging

Building notifications for use with Auto devices requires classes from the v4 support library. Use the Android SDK Manager to update the *Extras > Android Support Repository* **to version 9 or higher and the** *Extras > Android Support Library* **to version 21.0.2 or higher.**

After you have updated the support libraries, import them into your Android Studio development project by adding this dependency to your build.gradle file:

```
dependencies {
    ...
    compile 'com.android.support:support-v4:21.0.2'
}
```

For information about importing the support library into development projects for other development environments, see Support Library Setup.

Notify Users of Messages

A messaging app provides messages to a connected Auto dashboard using the notifications framework. When your messaging app has a message for a user, you build a specially configured notification that is received by the dashboard system and presented to the user. The Auto device manages the presentation on the dashboard screen and may play the message via text-to-speech. The dashboard system also handles voice interaction if the user replies to a message using verbal input.

The messaging user interface for Auto presents users with two levels of information about messages. The first level of notification tells users what *conversations* are available, and who they are with, but not the content of the messages. Typically, a conversation is one or more messages from another user to the Auto user.

The second level of the notification is the actual content of messages in the conversation. If a user indicates they want to hear the messages in a conversation, the Auto user interface plays the messages using text-to-speech.

This section describes how to notify Auto users that conversations are available and provide the content of messages in those conversations.

Create conversation read and reply intents

Unread conversation objects contain intents for reading and replying to a conversation. You create a `PendingIntent` object for each of these actions, so the Auto device can notify your app of action taken by the Auto user on a particular conversation.

The following example code demonstrates how to define a `PendingIntent` to let your app know if a conversation was read to the Auto user:

```
Intent msgHeardIntent = new Intent()
    .addFlags(Intent.FLAG_INCLUDE_STOPPED_PACKAGES)
    .setAction("com.myapp.messagingservice.MY_ACTION_MESSAGE_HEARD")
    .putExtra("conversation_id", thisConversationId);
```

```
PendingIntent msgHeardPendingIntent =
    PendingIntent.getBroadcast(getApplicationContext(),
        thisConversationId,
        msgHeardIntent,
        PendingIntent.FLAG_UPDATE_CURRENT);
```

In this example, `thisConversationId` is an integer that identifies the current conversation. The value of `Intent.setAction()` is the intent filter identifier for heard messages which you defined in your app manifest, as shown in Define read and reply intent filters.

If your app supports replying to conversations, you also create a `PendingIntent` for each conversation to notify your app that the user has replied. The following code example shows you how to build this intent for use with a particular conversation:

```
Intent msgReplyIntent = new Intent()
    .addFlags(Intent.FLAG_INCLUDE_STOPPED_PACKAGES)
    .setAction("com.myapp.messagingservice.MY_ACTION_MESSAGE_REPLY")
    .putExtra("conversation_id", thisConversationId);

PendingIntent msgReplyPendingIntent = PendingIntent.getBroadcast(
    getApplicationContext(),
    thisConversationId,
    msgReplyIntent,
    PendingIntent.FLAG_UPDATE_CURRENT);
```

Once again, `thisConversationId` is an integer that uniquely identifies this conversation, and the value you pass to `Intent.setAction()` is the intent filter identifier you defined for replies in your app manifest.

Set up the conversation builder

Messaging notifications for Auto organize messages into conversations using the `NotificationCompat.CarExtender.UnreadConversation` class, that represents an unread or new portion of a conversation from a particular sender. It contains a list of messages from the sender.

You generally do not configure the `UnreadConversation` directly. Instead, you configure an

UnreadConversation.Builder with the information about the conversation, as shown in the following example code.

```
// Build a RemoteInput for receiving voice input in a Car
Notification
RemoteInput remoteInput = new
RemoteInput.Builder(MY_VOICE_REPLY_KEY)

.setLabel(getApplicationContext().getString(R.string.notifi

        .build();

// Create an unread conversation object to organize a
group of messages
// from a particular sender.
UnreadConversation.Builder unreadConvBuilder =
    new UnreadConversation.Builder(conversationName)
        .setReadPendingIntent(msgHeardPendingIntent)
        .setReplyAction(msgReplyPendingIntent,
remoteInput);
```

Note: You won't actually create the UnreadConversation until you are almost ready to send the message.

This conversation object includes a PendingIntent, which allows the Auto device to signal your app that the conversation has been read by the Auto user. The construction of this intent is discussed in the Creating conversation read and reply intents section.

If your app supports replying to a conversation, you must call the UnreadConversation.Builder.setReplyAction() method and provide a pending intent to pass that user action back to your app. The UnreadConversation object you create must also include a RemoteInput object. When the Auto user receiving this conversation speaks a reply, the remote input objects lets your app get a text version of the voice reply.

Sending Messages

When a message arrives for a conversation, you take the following steps to dispatch it as a notification to Auto.

First, add the message to the `UnreadConversation.Builder` for this conversation, and update its timestamp:

```
unreadConvBuilder.addMessage(messageString)
    .setLatestTimestamp(currentTimestamp);
```

Note: If you are sending several messages at once, add them to the `UnreadConversation.Builder` in order, from oldest to newest.

Then create the `NotificationCompat.Builder` object that builds the actual notification. You need to use the pending intents you created in the previous step.

```
NotificationCompat.Builder notificationBuilder =
    new
NotificationCompat.Builder(getApplicationContext())
        .setSmallIcon(smallIconResourceID)
        .setLargeIcon(largeIconBitmap);
```

`smallIconResourceID`
: The resource ID of a small icon to use for the conversation. This is typically a generic icon for the messaging app.

`largeIconBitmap`
: A `Bitmap` of a large version of the icon. This is typically a conversation-specific graphic. For example, if this is a chat app, the large icon would be a picture of the person the user is chatting with.

`messageString`

The text of the message you want to send. (If you are sending several messages at once, concatenate them into a single string, with the oldest message first.)

`currentTimestamp`
: The message timestamp. (If you are sending several messages at once, use the timestamp of the most recent message.)

`conversationName`
: The name you chose for this conversation (for example, the name of the person the user is chatting with). This should be the same conversation name you used when you created the `UnreadConversation.Builder`.

`msgHeardPendingIntent`
: The pending intent object you created in Create conversation read and reply intents.

You'll also need to extend the `NotificationCompat.Builder` with the `CarExtender`. This is where you actually create the `UnreadConversation` object using the builder you just created, and attach it to the `CarExtender`:

```
notificationBuilder.extend(new CarExtender()
    .setUnreadConversation(unreadConvBuilder.build());
```

Note: If you wish, you can set an override icon or color for the `CarExtender` by calling `setLargeIcon()` or `setColor()`. The override icon or color is used when the notification is handled by a car, and has no effect if the notification is handled on the Android device. This is useful if the notification's default icon or color are not suitable for the car's display.

Once you've done all this, you use your app's `NotificationManagerCompat` to send the notification:

```
NotificationManagerCompat msgNotificationManager =
    NotificationManagerCompat.from(context);
msgNotificationManager.notify(notificationTag,
    notificationId, notificationBuilder.build());
```

Handle User Actions

When your create and dispatch a notification for messaging, you specify intents to be triggered when the Auto user hears the message and when the user dictates a reply. Your app indicates to the Android framework that it handles these intends by registering them through its manifest, as discussed in Define read and reply intent filters.

In addition to registering these intent filters, your app must provide code to handle these actions. Your app can do this by providing a service or `BroadcastReceiver` objects that handle these intents.

For more information about intents, see Intents and Intent Filters.

Handling a message heard action

When a user listens to a messaging conversation through the Auto user interface, the dashboard device sends a read intent based on how your app defined the messaging notification. Your app catches that intent and invokes the broadcast receiver class associated with it, or the service method set up to handle that action.

The following code example shows how to define a `BroadcastReceiver` class to handle a received message heard intent:

```
public class MyMessageHeardReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {

        // If you set up the intent as described in
        // "Create conversation read and reply intents",
        // you can get the conversation ID by calling:
        int thisConversationId =
```

```
intent.getIntExtra("conversation_id", -1);

        // Remove the notification to indicate it has been read
        // and update the list of unread conversations in your app.
    }
}
```

Once a notification is read, your app can remove it by calling `NotificationManagerCompat.cancel()` with the notification ID. Within your app, you should mark the messages provided in the notification as read.

Note: An alternative to this implementation is to use a service in a `PendingIntent`.

Handling a reply action

When a user replies to a messaging conversation through the Auto user interface, the dashboard system sends a reply intent based on how your app defined the messaging notification. Your app catches that intent and invokes the broadcast receiver class associated with it, or the service method set up to handle that action.

The following code example shows how to define a `BroadcastReceiver` class to handle a received message reply intent:

```
  public class MyMessageReplyReceiver extends BroadcastReceiver {


    @Override
    public void onReceive(Context context, Intent intent) {
        // If you set up the intent as described in
        // "Create conversation read and reply intents",
        // you can get the conversation ID by calling:
        int thisConversationId =
intent.getIntExtra("conversation_id", -1).

    }

    /**
     * Get the message text from the intent.
     * Note that you should call
     * RemoteInput.getResultsFromIntent() to process
     * the RemoteInput.
     */
    private CharSequence getMessageText(Intent intent) {
        Bundle remoteInput =
            RemoteInput.getResultsFromIntent(intent);
        if (remoteInput != null) {
            return remoteInput.getCharSequence(MY_VOICE_REPLY_KEY);
        }
```

```
        return null;
    }

}
```

Next Get started Next class

Building Apps for Work

The Android framework provides features to support the security, data separation, and administration needs of a enterprise environment. As an app developer, you can make your app more appealing to corporate customers by gracefully handling enterprise security and feature restrictions. You can also modify your app so that technology administrators can remotely configure it for use with enterprise resources.

To help businesses incorporate Android devices and apps into the workplace, Google provides the Android for Work program, which offers a suite of APIs and services for device distribution and administration. Through this program companies can connect with Enterprise Mobility Management (EMM) providers to help integrate Android with their businesses.

For more information, follow the links below to learn how to update your Android app to support the enterprise environment or build enterprise-specific solutions.

App Development for Enterprises

Learn how to make your app function smoothly in corporate environments that restrict device features and data access. Go further to

support enterprise use of your app by enabling restrictions that corporate technology administrators can use to remotely configure your app:

Device and App Administration

Learn how to build policy controller apps that enable enterprise technology administrators to manage devices, manage corporate apps, and provide access to company resources:

Next Get started Next class

Ensuring Compatibility with Managed Profiles

Previous Next Get started

This lesson teaches you to

1. Prevent Failed Intents
2. Share Files Across Profiles
3. Test your App for Compatibility with Managed Profiles

Resources

- BasicManagedProfile

The Android platform allows devices to have managed profiles. A managed profile is controlled by an administrator, and the functionality available to it is set separately from the functionality of the user's primary profile. This approach lets enterprises control the environment where company-specific apps and data are running on a user's device, while still letting users use their personal apps and profiles.

This lesson shows you how to modify your application so it functions reliably on a device with managed profiles. You don't need to do anything besides the ordinary app-development best practices. However, some of these best practices become especially important on devices with managed profiles. This document highlights the issues you need to be aware of.

Overview

Users often want to use their personal devices in an enterprise setting.

This situation can present enterprises with a dilemma. If the user can use their own device, the enterprise has to worry that confidential information (like employee emails and contacts) are on a device the enterprise does not control.

To address this situation, Android 5.0 (API level 21) allows enterprises to set up *managed profiles*. If a device has a managed profile, the profile's settings are under the control of the enterprise administrator. The administrator can choose which apps are allowed for that profile, and can control just what device features are available to the profile.

If a device has a managed profile, there are implications for apps running on the device, no matter which profile the app is running under:

- By default, most intents do not cross from one profile to the other. If an app running on profile fires an intent, there is no handler for the intent on that profile, and the intent is not allowed to cross to the other profile due to profile restrictions, the request fails and the app may shut down unexpectedly.
- The profile administrator can limit which system apps are available on the managed profile. This restriction can also result in there being no handler for some common intents on the managed profile.
- Since the managed and unmanaged profiles have separate storage areas, a file URI that is valid on one profile is not valid on the other. Any intent fired on one profile might be handled on the other (depending on profile settings), so it is not safe to attach file URIs to intents.

Prevent Failed Intents

On a device with a managed profile, there are restrictions on whether intents can cross from one profile to another. In most cases, when an intent is fired off, it is handled on the same profile where it is fired. If there is no handler for the intent *on that profile*, the intent is not handled and the app that fired it may shut down unexpectedly—even if there's a handler for the intent on the other profile.

The profile administrator can choose which intents are allowed to cross from one profile to another. Since the administrator makes this decision, there's no way for you to know in advance *which* intents are allowed to cross this boundary. The administrator sets this policy, and is free to change it at any time.

Before your app starts an activity, you should verify that there is a suitable resolution. You can verify that there is an acceptable resolution by calling `Intent.resolveActivity()`. If there is no way to resolve the intent, the method returns `null`. If the method returns non-null, there is at least one way to resolve the intent, and it is safe to fire off the intent. In this case, the intent could be resolvable either because there is a handler on the current profile, or because the intent is allowed to cross to a handler on the other profile. (For more information about resolving intents, see Common Intents.)

For example, if your app needs to set timers, it would need to check that there's a valid handler for the `ACTION_SET_TIMER` intent. If the app cannot resolve the intent, it should take an appropriate action (such as showing an error message).

```
public void startTimer(String message, int seconds) {

    // Build the "set timer" intent
    Intent timerIntent = new Intent(AlarmClock.ACTION_SET_TIMER)
            .putExtra(AlarmClock.EXTRA_MESSAGE, message)
            .putExtra(AlarmClock.EXTRA_LENGTH, seconds)
            .putExtra(AlarmClock.EXTRA_SKIP_UI, true);

    // Check if there's a handler for the intent
    if (timerIntent.resolveActivity(getPackageManager()) == null) {

        // Can't resolve the intent! Fail this operation cleanly
        // (perhaps by showing an error message)

    } else {
        // Intent resolves, it's safe to fire it off
        startActivity(timerIntent);

    }
}
```

Share Files Across Profiles

Sometimes an app needs to provide other apps with access to its own

files. For example, an image gallery app might want to share its images with image editors. There are two ways you would ordinarily share a file: with a *file URI* or a *content URI*.

A file URI begins with the `file:` prefix, followed by the absolute path of the file on the device's storage. However, because the managed profile and the personal profile use separate storage areas, a file URI that is valid on one profile is not valid on the other. This situation means that if you attach a file URI to an intent, and the intent is handled on the other profile, the handler is not able to access the file.

Instead, you should share files with *content URIs*. Content URIs identify the file in a more secure, shareable fashion. The content URI contains the file path, but also the authority that provides the file, and an ID number identifying the file. You can generate a content ID for any file by using a `FileProvider`. You can then share that content ID with other apps (even on the other profile). The recipient can use the content ID to get access to the actual file.

For example, here's how you would get the content URI for a specific file URI:

```
// Open File object from its file URI
File fileToShare = new File(fileUriToShare);

Uri contentUriToShare = FileProvider.getUriForFile(getContext(),
        "com.example.myapp.fileprovider", fileToShare);
```

When you call the `getUriForFile()` method, you must include the file provider's authority (in this example, `"com.example.myapp.fileprovider"`), which is specified in the `<provider>` element of your app manifest. For more information about sharing files with content URIs, see Sharing Files.

Test your App for Compatibility with Managed Profiles

You should test your app in a managed-profile environment to catch problems that would cause your app to fail on a device with managed profiles. In particular, testing on a managed-profile device is a good way

to make sure that your app handles intents properly: not firing intents that can't be handled, not attaching URIs that don't work cross-profile, and so on.

We have provided a sample app, BasicManagedProfile, which you can use to set up a managed profile on an Android device that runs Android 5.0 (API level 21) and higher. This app offers you a simple way to test your app in a managed-profile environment. You can also use this app to configure the managed profile as follows:

- Specify which default apps are available on the managed profile
- Configure which intents are allowed to cross from one profile to the other

If you manually install an app over a USB cable to a device which has a managed profile, the app is installed on both the managed and the unmanaged profile. Once you have installed the app, you can test the app under the following conditions:

- If an intent would ordinarily be handled by a default app (for example, the camera app), try disabling that default app on the managed profile, and verify that the app handles this appropriately.
- If you fire an intent expecting it to be handled by some other app, try enabling and disabling that intent's permission to cross from one profile to another. Verify that the app behaves properly under both circumstances. If the intent is not allowed to cross between profiles, verify the app's behavior both when there is a suitable handler on the app's profile, and when there is not. For example, if your app fires a map-related intent, try each of the following scenarios:
    - The device allows map intents to cross from one profile to the other, and there is a suitable handler on the other profile (the profile the app is not running on)
    - The device does not allow map intents to cross between profiles, but there is a suitable handler on the app's profile
    - The device does not allow map intents to cross between profiles, and there is no suitable handler for map intents on the device's profile
- If you attach content to an intent, verify that the intent behaves

properly both when it is handled on the app's profile, and when it crosses between profiles.

Testing on managed profiles: Tips and tricks

There are a few tricks that you may find helpful in testing on a managed-profile device.

- As noted, when you side-load an app on a managed profile device, it is installed on both profiles. If you wish, you can delete the app from one profile and leave it on the other.
- Most of the activity manager commands available in the Android Debug Bridge (adb) shell support the `--user` flag, which lets you specify which user to run as. By specifying a user, you can choose whether to run as the unmanaged or managed profile. For more information, see ADB Shell Commands.
- To find the active users on a device, use the adb package manager's `list users` command. The first number in the output string is the user ID, which you can use with the `--user` flag. For more information, see ADB Shell Commands.

For example, to find the users on a device, you would run this command:

```
$ adb shell pm list users
UserInfo{0:Drew:13} running
UserInfo{10:Work profile:30} running
```

In this case, the unmanaged profile ("Drew") has the user ID 0, and the managed profile has the user ID 10. To run an app in the work profile, you would use a command like this:

```
$ adb shell am start --user 10 \
-n "com.example.myapp/com.example.myapp.testactivity" \
-a android.intent.action.MAIN -c android.intent.category.LAUNCHER
```

Next Get started Next class

Implementing App Restrictions

Previous Next Get started

This lesson teaches you to

1. Define App Restrictions
2. Check App Restrictions
3. Listen for App Restriction Changes

Resources

- AppRestrictionSchema sample app
- AppRestrictionEnforcer sample app

If you are developing apps for the enterprise market, you may need to satisfy particular requirements set by a company's policies. Application restrictions allow the enterprise administrator to remotely specify settings for apps. This capability is particularly useful for enterprise-approved apps deployed to a managed profile.

For example, an enterprise might require that approved apps allow the enterprise administrator to:

- Whitelist or blacklist URLs for a web browser
- Configure whether an app is allowed to sync content via cellular, or just by Wi-Fi
- Configure the app's email settings

This guide shows how to implement these configuration settings in your app.

Note: For historical reasons, these configuration settings are known as *restrictions,* and are implemented with files and classes that use this

term (such as `RestrictionsManager`). However, these restrictions can actually implement a wide range of configuration options, not just restrictions on app functionality.

Remote Configuration Overview

Apps define the restrictions and configuration options that can be remotely set by an administrator. These restrictions are arbitrary configuration settings that can be changed by a restrictions provider. If your app is running on an enterprise device's managed profile, the enterprise administrator can change your app's restrictions.

The restrictions provider is another app running on the same device. This app is typically controlled by the enterprise administrator. The enterprise administrator communicates restriction changes to the restrictions provider app. That app, in turn, changes the restrictions on your app.

To provide externally configurable restrictions:

- Declare the restrictions in your app manifest. Doing so allows the enterprise administrator to read the app's restrictions through Google Play APIs.
- Whenever the app resumes, use the `RestrictionsManager` object to check the current restrictions, and change your app's UI and behavior to conform with those restrictions.
- Listen for the `ACTION_APPLICATION_RESTRICTIONS_CHANGED` intent. When you receive this broadcast, check the `RestrictionsManager` to see what the current restrictions are, and make any necessary changes to your app's behavior.

Define App Restrictions

Your app can support any restrictions you want to define. You declare the app's restrictions in a *restrictions file*, and declare the restrictions file in the manifest. Creating a restrictions file allows other apps to examine the restrictions your app provides. Enterprise Mobility Management (EMM) partners can read your app's restrictions by using Google Play

APIs.

To define your app's remote configuration options, put the following element in your manifest's `<application>` element:

```
<meta-data android:name="android.content.APP_RESTRICTIONS"
    android:resource="@xml/app_restrictions" />
```

Create a file named `app_restrictions.xml` in your app's `res/xml` directory. The structure of that file is described in the reference for `RestrictionsManager`. The file has a single top-level `<restrictions>` element, which contains one `<restriction>` child element for every configuration option the app has.

Note: Do not create localized versions of the restrictions file. Your app is only allowed to have a single restrictions file, so restrictions will be consistent for your app in all locales.

In an enterprise environment, an EMM will typically use the restrictions schema to generate a remote console for IT administrators, so the administrators can remotely configure your application.

For example, suppose your app can be remotely configured to allow or forbid it to download data over a cellular connection. Your app could have a `<restriction>` element like this:

```
<?xml version="1.0" encoding="utf-8"?>
<restrictions
xmlns:android="http://schemas.android.com/apk/res/android" >

  <restriction
    android:key="downloadOnCellular"
    android:title="App is allowed to download data via cellular"
    android:restrictionType="bool"
    android:description="If 'false', app can only download data via
Wi-Fi"
    android:defaultValue="true" />

</restrictions>
```

The supported types for the `android:restrictionType` element are documented in the reference for `RestrictionsManager`.

You use each restriction's `android:key` attribute to read its value from a restrictions bundle. For this reason, each restriction must have a unique

key string, and the string *cannot* be localized. It must be specified with a string literal.

Note: In a production app, `android:title` and `android:description` should be drawn from a localized resource file, as described in Localizing with Resources.

The restrictions provider can query the app to find details on the app's available restrictions, including their description text. Restrictions providers and enterprise administrators can change your app's restrictions at any time, even when the app is not running.

Check App Restrictions

Your app is not automatically notified when other apps change its restriction settings. Instead, you need to check what the restrictions are when your app starts or resumes, and listen for a system intent to find out if the restrictions change while your app is running.

To find out the current restriction settings, your app uses a `RestrictionsManager` object. Your app should check for the current restrictions at the following times:

- When the app starts or resumes, in its `onResume()` method
- When the app is notified of a restriction change, as described in Listen for Device Configuration Changes

To get a `RestrictionsManager` object, get the current activity with `getActivity()`, then call that activity's `Activity.getSystemService()` method:

```
RestrictionsManager myRestrictionsMgr =
    (RestrictionsManager) getActivity()
        .getSystemService(Context.RESTRICTIONS_SERVICE);
```

Once you have a `RestrictionsManager`, you can get the current restrictions settings by calling its `getApplicationRestrictions()` method:

```
Bundle appRestrictions =
myRestrictionsMgr.getApplicationRestrictions();
```

Note: For convenience, you can also fetch the current restrictions with a `UserManager`, by calling `UserManager.getApplicationRestrictions()`. This method behaves exactly the same as `RestrictionsManager.getApplicationRestrictions()`.

The `getApplicationRestrictions()` method requires reading from data storage, so it should be done sparingly. Do not call this method every time you need to know the current restrictions. Instead, you should call it once when your app starts or resumes, and cache the fetched restrictions bundle. Then listen for the `ACTION_APPLICATION_RESTRICTIONS_CHANGED` intent to find out if restrictions change while your app is active, as described in Listen for Device Configuration Changes.

Reading and applying restrictions

The `getApplicationRestrictions()` method returns a `Bundle` containing a key-value pair for each restriction that has been set. The values are all of type `Boolean`, `int`, `String`, and `String[]`. Once you have the restrictions `Bundle`, you can check the current restrictions settings with the standard `Bundle` methods for those data types, such as `getBoolean()` or `getString()`.

Note: The restrictions `Bundle` contains one item for every restriction that has been explicitly set by a restrictions provider. However, you *cannot* assume that a restriction will be present in the bundle just because you defined a default value in the restrictions XML file.

It is up to your app to take appropriate action based on the current restrictions settings. For example, if your app has a restriction specifying whether it can download data over a cellular connection, and you find that the restriction is set to `false`, you would have to disable data download except when the device has a Wi-Fi connection, as shown in the following example code:

```
boolean appCanUseCellular;

if appRestrictions.containsKey("downloadOnCellular") {
    appCanUseCellular =
appRestrictions.getBoolean("downloadOnCellular");
} else {
```

```
    // here, cellularDefault is a boolean set with the restriction's
    // default value
    appCanUseCellular = cellularDefault;
}

if (!appCanUseCellular) {
    // ...turn off app's cellular-download functionality
    // ...show appropriate notices to user
}
```

Listen for App Restriction Changes

Whenever an app's restrictions are changed, the system fires the ACTION_APPLICATION_RESTRICTIONS_CHANGED intent. Your app has to listen for this intent so you can change the app's behavior when the restriction settings change.

Note: The ACTION_APPLICATION_RESTRICTIONS_CHANGED intent is sent only to listeners that are dynamically registered, *not* to listeners that are declared in the app manifest.

The following code shows how to dynamically register a broadcast receiver for this intent:

```
IntentFilter restrictionsFilter =
    new IntentFilter(Intent.ACTION_APPLICATION_RESTRICTIONS_CHANGED);

BroadcastReceiver restrictionsReceiver = new BroadcastReceiver() {
  @Override public void onReceive(Context context, Intent intent) {

    // Get the current restrictions bundle
    Bundle appRestrictions =

    myRestrictionsMgr.getApplicationRestrictions();

    // Check current restrictions settings, change your app's UI and
    // functionality as necessary.

  }

};

registerReceiver(restrictionsReceiver, restrictionsFilter);
```

Note: Ordinarily, your app does not need to be notified about restriction changes when it is paused. Instead, you should unregister your broadcast receiver when the app is paused. When the app resumes, you first check for the current restrictions (as discussed in Check Device Restrictions),

then register your broadcast receiver to make sure you're notified about restriction changes that happen while the app is active.

Next Get started Next class

Building a Work Policy Controller

Previous Next Get started

This lesson teaches you to

1. Create a Managed Profile
2. Set Up Device Policies
3. Apply App Restrictions

You should also read

- Device Administration

Resources

- BasicManagedProfile
- AppRestrictionEnforcer

In an Android for Work deployment, an enterprise needs to maintain control over certain aspects of the employees' devices. The enterprise needs to ensure that work-related information is encrypted and is kept separate from employees' personal data. The enterprise may also need to limit device capabilities, such as whether the device is allowed to use its camera. And the enterprise may require that approved apps provide app restrictions, so the enterprise can turn app capability on or off as needed.

To handle these tasks, an enterprise develops and deploys a Work Policy Controller app. This app is installed on each employee's device. The controller app installed on each employee's device and creates a work user profile, which accesses enterprise apps and data separately from the user's personal account. The controller app also acts as the bridge

between the enterprise's management software and the device; the enterprise tells the controller app when it needs to make configuration changes, and the controller app makes the appropriate settings changes for the device and for other apps.

This lesson describes how to develop a Work Policy Controller app for devices in an Android for Work deployment. The lesson describes how to create a work user profile, how to set device policies, and how to apply restrictions to other apps running on the managed profile.

Note: This lesson does not cover the situation where the only profile on the device is the managed profile, under the enterprise's control.

Device Administration Overview

In an Android for Work deployment, the enterprise administrator can set policies to control the behavior of employees' devices and apps. The enterprise administrator sets these policies with software provided by their Enterprise Mobility Management (EMM) provider. The EMM software communicates with a Work Policy Controller on each device. The Work Policy Controller, in turn, manages the settings and behavior of the work user profile on each individual's device.

Note: A Work Policy Controller is built on the existing model used for device administration applications, as described in Device Administration. In particular, your app needs to create a subclass of `DeviceAdminReceiver`, as described in that document.

Managed profiles

Users often want to use their personal devices in an enterprise setting. This situation can present enterprises with a dilemma. If the user can use their own device, the enterprise has to worry that confidential information (like employee emails and contacts) are on a device the enterprise does not control.

To address this situation, Android 5.0 (API level 21) allows enterprises to

set up a special work user profile using the Managed Profile API. This user profile is called a *managed profile*, or a *work profile* in the Android for Work program. If a device has a managed profile for work, the profile's settings are under the control of the enterprise administrator. The administrator can choose which apps are allowed for that profile, and can control just what device features are available to the profile.

Create a Managed Profile

To create a managed profile on a device that already has a personal profile, first check that the device can support a managed profile, by seeing if the device supports the FEATURE_MANAGED_USERS system feature:

```
PackageManager pm = getPackageManager();
if (!pm.hasSystemFeature(PackageManager.FEATURE_MANAGED_USERS)) {

    // This device does not support native managed profiles!

}
```

If the device supports managed profiles, create one by sending an intent with an ACTION_PROVISION_MANAGED_PROFILE action. Include the device admin package name as an extra.

```
Activity provisioningActivity = getActivity();

// You'll need the package name for the WPC app.
String myWPCPackageName = "com.example.myWPCApp";

// Set up the provisioning intent
Intent provisioningIntent =
        new Intent("android.app.action.PROVISION_MANAGED_PROFILE");
intent.putExtra(myWPCPackageName,

provisioningActivity.getApplicationContext().getPackageName());

if
(provisioningIntent.resolveActivity(provisioningActivity.getPackageMana

         == null) {

    // No handler for intent! Can't provision this device.
    // Show an error message and cancel.
} else {

    // REQUEST_PROVISION_MANAGED_PROFILE is defined
    // to be a suitable request code
    startActivityForResult(provisioningIntent,
            REQUEST_PROVISION_MANAGED_PROFILE);
    provisioningActivity.finish();
```

```
}
```

The system responds to this intent by doing the following:

- Verifies that the device is encrypted. If it is not, the system prompts the user to encrypt the device before proceeding.
- Creates a managed profile.
- Removes non-required applications from the managed profile.
- Copies the Work Policy Controller application into the managed profile and sets it as the profile owner.

Override `onActivityResult()` to see whether the provisioning was successful, as shown in the following example code:

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

    // Check if this is the result of the provisioning activity
    if (requestCode == REQUEST_PROVISION_MANAGED_PROFILE) {

        // If provisioning was successful, the result code is
        // Activity.RESULT_OK
        if (resultCode == Activity.RESULT_OK) {
            // Hurray! Managed profile created and provisioned!
        } else {
            // Boo! Provisioning failed!
        }
        return;

    } else {
        // This is the result of some other activity, call the
superclass
        super.onActivityResult(requestCode, resultCode, data);
    }
}
```

After Creating the Managed Profile

When the profile has been provisioned, the system calls the Work Policy Controller app's `DeviceAdminReceiver.onProfileProvisioningComplete()` method. Override this callback method to finish enabling the managed profile.

Typically, your `DeviceAdminReceiver.onProfileProvisioningComplete()` callback implementation would perform these tasks:

- Verify that the device is complying with the EMM's device policies, as described in Set Up Device Policies
- Enable any system applications that the administrator chooses to make available within the managed profile, using `DevicePolicyManager.enableSystemApp()`
- If the device uses Google Play for Work, add the Google account to the managed profile with `AccountManager.addAccount()`, so administrators can install applications to the device

Once you have completed these tasks, call the device policy manager's `setProfileEnabled()` method to activate the managed profile:

```
// Get the device policy manager
DevicePolicyManager myDevicePolicyMgr =
        (DevicePolicyManager)
getSystemService(Context.DEVICE_POLICY_SERVICE);

ComponentName componentName =
myDeviceAdminReceiver.getComponentName(this);

// Set the name for the newly created managed profile.
myDevicePolicyMgr.setProfileName(componentName, "My New Managed
Profile");

// ...and enable the profile
manager.setProfileEnabled(componentName);
```

Set Up Device Policies

The Work Policy Controller app is responsible for applying the enterprise's device policies. For example, a particular enterprise might require that all devices become locked after a certain number of failed attempts to enter the device password. The controller app queries the EMM to find out what the current policies are, then uses the Device Administration API to apply those policies.

For information on how to apply device policies, see the Device Administration guide.

Apply App Restrictions

Enterprise environments may require that approved apps implement apps

implement security or feature restrictions. App developers must implement these restrictions and declare them for use by enterprise administrators, as described in Implementing App Restrictions. The Work Policy Controller receives restriction changes from the enterprise administrator, and forwards those restriction changes to the apps.

For example, a particular news app might have a restriction setting that controls whether the app is allowed to download videos over a cellular network. When the EMM wants to disable cellular downloads, it sends a notification to the controller app. The controller app, in turn, notifies the news app that the restriction setting has changed.

Note: This document covers how the Work Policy Controller app changes the restriction settings for the other apps on the managed profile. Details on how the Work Policy Controller app communicates with the EMM are out of scope for this document.

To change an app's restrictions, call the `DevicePolicyManager.setApplicationRestrictions()` method. This method is passed three parameters: the controller app's `DeviceAdminReceiver`, the package name of the app whose restrictions are being changed, and a `Bundle` that contains the restrictions you want to set.

For example, suppose there's an app on the managed profile with the package name `"com.example.newsfetcher"`. This app has a single boolean restriction that can be configured, with the key `"downloadByCellular"`. If this restriction is set to `false`, the newsfetcher app is not allowed to download data through a cellular network; it must use a Wi-Fi network instead.

If your Work Policy Controller app needs to turn off cellular downloads, it would first fetch the device policy service object, as described above. It then assembles a restrictions bundle and passes this bundle to `setApplicationRestrictions()`:

```
// Fetch the DevicePolicyManager
DevicePolicyManager myDevicePolicyMgr =
        (DevicePolicyManager) thisActivity
                .getSystemService(Context.DEVICE_POLICY_SERVICE);
```

```
// Set up the restrictions bundle
bundle restrictionsBundle = new Bundle();
restrictionsBundle.putBoolean("downloadByCellular", false);

// Pass the restrictions to the policy manager. Assume the WPC app
// already has a DeviceAdminReceiver defined
(myDeviceAdminReceiver).
myDevicePolicyMgr.setApplicationRestrictions(
        myDeviceAdminReceiver, "com.example.newsfetcher",
restrictionsBundle);
```

Note: The device policy service conveys the restrictions change to the app you name. However, it is up to that app to actually implement the restriction. For example, in this case, the app would be responsible for disabling its ability to use cellular networks for video downloads. Setting the restriction does not cause the system to enforce this restriction on the app. For more information, see Implementing App Restrictions.

Next Get started Next class

Best Practices for Interaction and Engagement

These classes teach you how to engage and retain your users by implementing the best interaction patterns for Android. For instance, to help users quickly discover content in your app, your app should match their expectations for user interaction on Android. And to keep your users coming back, you should take advantage of platform capabilities that reveal and open your content without requiring users to go through the app launcher.

Designing Effective Navigation

Previous Next Get started

Dependencies and prerequisites

This class is not specific to any particular version of the Android platform. It is also primarily design-focused and does not require knowledge of the Android SDK. That said, you should have experience using an Android device for a better understanding of the context in which Android applications run.

You should also have basic familiarity with the Action Bar (pattern docs at Android Design), used across most applications in devices running Android 3.0 and later.

One of the very first steps to designing and developing an Android application is to determine what users are able to see and do with the app. Once you know what kinds of data users are interacting with in the app, the next step is to design the interactions that allow users to navigate across, into, and back out from the different pieces of content within the app.

This class shows you how to plan out the high-level screen hierarchy for your application and then choose appropriate forms of navigation to allow users to effectively and intuitively traverse your content. Each lesson covers various stages in the interaction design process for navigation in Android applications, in roughly chronological order. After going through the lessons in this class, you should be able to apply the methodology and navigation paradigms outlined here to your own applications, providing a coherent navigation experience for your users.

Lessons

Planning Screens and Their Relationships

Learn how to choose which screens your application should contain. Also learn how to choose which screens should be directly reachable from others. This lesson introduces a hypothetical news application to serve as an example for later lessons.

Planning for Multiple Touchscreen Sizes

Learn how to group related screens together on larger-screen devices to optimize use of available screen space.

Providing Descendant and Lateral Navigation

Learn about techniques for allowing users to navigate deep into, as well as across, your content hierarchy. Also learn about pros and cons of, and best practices for, specific navigational UI elements for various situations.

Providing Ancestral and Temporal Navigation

Learn how to allow users to navigate upwards in the content hierarchy. Also learn about best practices for the *Back* button and temporal navigation, or navigation to previous screens that may not be hierarchically related.

Putting it All Together: Wireframing the Example App

Learn how to create screen wireframes (low-fidelity graphic mockups) representing the screens in a news application based on the desired information model. These wireframes utilize navigational elements discussed in previous lessons to demonstrate intuitive and efficient navigation.

Next Get started Next class

Planning Screens and Their Relationships

Previous Next Get started

This lesson teaches you to

1. Create a Screen List
2. Diagram Screen Relationships
3. Go Beyond a Simplistic Design

Most apps have an inherent information model that can be expressed as a tree or graph of object types. In more obvious terms, you can draw a diagram of different kinds of information that represents the types of things users interact with in your app. Software engineers and data architects often use entity-relationship diagrams (ERDs) to describe an application's information model.

Let's consider an example application that allows users to browse through a set of categorized news stories and photos. One possible model for such an app is shown below in the form of an ERD.

Figure 1. Entity-relationship diagram for the example news application.

Create a Screen List

Once you define the information model, you can begin to define the contexts necessary to enable users to effectively discover, view, and act upon the data in your application. In practice, one way to do this is to *determine the exhaustive set of screens* needed to allow users to navigate to and interact with the data. The set of screens we actually expose should generally vary depending on the target device; it's important to consider this early in the design process to ensure that the application can adapt to its environment.

In our example application, we want to enable users to **view**, **save**, and **share** *categorized* **stories** and **photos**. Below is an exhaustive list of screens that covers these use cases.

- Home or "launchpad" screen for accessing stories and photos
- List of categories
- List of news stories for a given category
- Story detail view (from which we can save and share)
- List of photos, uncategorized
- Photo detail view (from which we can save and share)
- List of all saved items
- List of saved photos
- List of saved stories

Diagram Screen Relationships

Now we can define the directed relationships between screens; an arrow from one screen *A* to another screen *B* implies that screen *B* should be directly reachable via some user interaction in screen *A*. Once we define both the set of screens and the relationships between them, we can express these in concert as a screen map, which shows all of your screens and their relationships:

Figure 2. Exhaustive screen map for the example news application.

If we later wanted to allow users to submit news stories or upload photos, we could add additional screens to this diagram.

Go Beyond a Simplistic Design

At this point, it's possible to design a completely functional application from this exhaustive screen map. A simplistic user interface could consist of lists and buttons leading to child screens:

- Buttons leading to different sections (e.g., stories, photos, saved items)
- Vertical lists representing collections (e.g., story lists, photo lists, etc.)
- Detail information (e.g., story view, full-screen photo view, etc.)

However, you can use screen grouping techniques and more sophisticated navigation elements to present content in a more intuitive and device-sensitive way. In the next lesson, we explore screen grouping techniques, such as providing multi-pane layouts for tablet devices. Later, we'll dive into the various navigation patterns common on Android.

Next Get started Next class

Planning for Multiple Touchscreen Sizes

Previous Next Get started

This lesson teaches you to

1. Group Screens with Multi-pane Layouts
2. Design for Multiple Tablet Orientations
3. Group Screens in the Screen Map

You should also read

- Android Design: Multi-pane Layouts
- Designing for Multiple Screens

The exhaustive screen map from the previous lesson isn't tied to a particular device form factor, although it can generally look and work okay on a handset or similar-size device. But Android applications need to adapt to a number of different types of devices, from 3" handsets to 10" tablets to 42" TVs. In this lesson we explore reasons and tactics for grouping together multiple screens from the exhaustive map.

Note: Designing applications for television sets also requires attention to other factors, including interaction methods (i.e., the lack of a touch screen), legibility of text at large reading distances, and more. Although this discussion is outside the scope of this class, you can find more information on designing for TVs in the Google TV documentation for design patterns.

Group Screens with Multi-pane Layouts

Multi-pane Layout Design

For design guidelines, read Android Design's Multi-pane Layouts pattern guide.

3 to 4-inch screens are generally only suitable for showing a single vertical pane of content at a time, be it a list of items, or detail information about an item, etc.

Thus on such devices, screens generally map one-to-one with levels in the information hierarchy (*categories → object list → object detail*).

Larger screens such as those found on tablets and TVs, on the other hand, generally have much more available screen space and are able to present multiple panes of content. In landscape, panes are usually ordered from left to right in increasing detail order. Users are especially accustomed to multiple panes on larger screens from years and years of desktop application and desktop web site use. Many desktop applications and websites offer a left-hand navigation pane or use a master/detail two-pane layout.

In addition to addressing these user expectations, it's usually necessary to provide multiple panes of information on tablets to avoid leaving too much whitespace or unwittingly introducing awkward interactions, for example 10 x 0.5-inch buttons.

The following figures demonstrate some of the problems that can arise when moving a UI (user interface) design into a larger layout and how to address these issues with multi-pane layouts:

Figure 1. Single pane layouts on large screens in landscape lead to awkward whitespace and exceedingly long line lengths.

Figure 2. Multi-pane layouts in landscape result in a better visual balance while offering more utility and legibility.

Implementation Note: After deciding on the screen size at which to draw the line between single-pane and multi-pane layouts, you can provide different layouts containing one or multiple panes for devices in varying screen size buckets (such as `large`/`xlarge`) or varying minimum screen widths (such as `sw600dp`).

Implementation Note: While a single screen is implemented as an `Activity` subclass, individual content panes can be implemented as `Fragment` subclasses. This maximizes code re-use across different form factors and across screens that share content.

Design for Multiple Tablet Orientations

Although we haven't begun arranging user interface elements on our screens yet, this is a good time to consider how your multi-pane screens will adapt to different device orientations. Multi-pane layouts in landscape work quite well because of the large amount of available horizontal space. However, in the portrait orientation,

your horizontal space is more limited, so you may need to design a separate layout for this orientation.

Below are a few common strategies for creating portrait tablet layouts.

- **Stretch**

  The most straightforward strategy is to simply stretch each pane's width to best present the content in each pane in the portrait orientation. Panes could have fixed widths or take a certain percentage of the available screen width.

- **Expand / collapse**

  A variation on the stretch strategy is to collapse the contents of the left pane when in portrait. This works quite well with master/detail panes where the left (master) pane contains easily collapsible list items. An example would be for a realtime chat application. In landscape, the left list could contain chat contact photos, names, and online statuses. In portrait, horizontal space could be collapsed by hiding contact names and only showing photos and online status indicator icons. Optionally also provide an expand control that allows the user to expand the left pane content to its larger width and vice versa.

- **Show / Hide**

  In this scenario, the left pane is completely hidden in portrait mode. However, *to ensure the functional parity* of your screen in portrait and landscape, the left pane should be made available via an onscreen affordance (such as a button). It's usually appropriate to use the *Up* button in the Action Bar (pattern docs at Android Design) to show the left pane, as is discussed in a later lesson.

- **Stack**

The last strategy is to vertically stack your normally horizontally-arranged panes in portrait. This strategy works well when your panes aren't simple text-based lists, or when there are multiple blocks of content running along the primary content pane. Be careful to avoid the awkward whitespace problem discussed above when using this strategy.

Group Screens in the Screen Map

Now that we are able to group individual screens together by providing multi-pane layouts on larger-screen devices, let's apply this technique to our exhaustive screen map from the previous lesson to get a better sense of our application's hierarchy on such devices:

Figure 3. Updated example news application screen map for tablets.

In the next lesson we discuss *descendant* and *lateral* navigation, and explore more ways of grouping screens to maximize the intuitiveness and speed of content access in the application's user interface.

Next Get started Next class

Providing Descendant and Lateral Navigation

Previous Next Get started

This lesson teaches you about:

1. Buttons and Simple Targets
2. Lists, Grids, Carousels, and Stacks
3. Tabs
4. Horizontal Paging (Swipe Views)

You should also read

- Android Design: Buttons
- Android Design: Lists
- Android Design: Grid Lists
- Android Design: Tabs
- Android Design: Swipe Views

One way of providing access to the full range of an application's screens is to expose hierarchical navigation. In this lesson we discuss *descendant navigation*, allowing users to descend 'down' a screen hierarchy into a child screen, and *lateral navigation*, allowing users to access sibling screens.

Figure 1. Descendant and lateral navigation.

There are two types of sibling screens: collection-related and section-related screens. *Collection-related* screens represent individual items in the collection represented by the parent. *Section-related* screens represent different sections of information about the parent. For example, one section may show textual information about an object while another may provide a map of the object's geographic location. The number of section-related screens for a given parent is generally small.

Figure 2. Collection-related children and section-related children.

Descendant and lateral navigation can be provided using lists, tabs, and other user interface patterns. *User interface patterns*, much like software design patterns, are generalized, common solutions to recurring interaction design problems. We explore a few common lateral navigation patterns in the sections below.

Buttons and Simple Targets

Button Design

For design guidelines, read Android Design's Buttons guide.

For section-related screens, offering touchable and keyboard-focusable targets in the parent is generally the most straightforward and familiar kind of touch-based navigation interface. Examples of such targets include buttons, fixed-size list views, or text links, although the latter is not an ideal UI (user interface) element for touch-based navigation. Upon selecting one of these targets, the child screen is opened, replacing the current context (screen) entirely. Buttons and other simple targets are rarely used for representing items in a collection.

Figure 3. Example button-based navigation interface with relevant screen map excerpt. Also shows dashboard pattern discussed below.

A common, button-based pattern for accessing different top-level

application sections, is the dashboard pattern. A *dashboard* is a grid of large, iconic buttons that constitutes the entirety, or most of, the parent screen. The grid generally has either 2 or 3 rows and columns, depending on the number of top-level sections in the app. This pattern is a great way to present all the sections of the app in a visually rich way. The large touch targets also make this UI very easy to use. Dashboards are best used when each section is equally important, as determined by product decisions or better yet, real-world usage. However, this pattern doesn't visually work well on larger screens, and requires users to take an extra step to jump directly into the app's content.

More sophisticated user interfaces can make use of a variety of other user interaction patterns to improve content immediacy and presentation uniqueness, all the while remaining intuitive.

Lists, Grids, Carousels, and Stacks

List and Grid List Design

For design guidelines, read Android Design's Lists and Grid Lists guides.

For collection-related screens, and especially for textual information, vertically scrolling lists are often the most straightforward and familiar kind of interface. For more visual or media-rich content items such as photos or videos, vertically scrolling grids of items, horizontally scrolling lists (sometimes referred to as *carousels*), or stacks (sometimes referred to as *cards*) can be used instead. These UI elements are generally best used for presenting item collections or large sets of child screens (for example, a list of stories or a list of 10 or more news topics), rather than a small set of unrelated, sibling child screens.

Figure 4. Example list-, grid-, and carousel-based navigation interfaces with relevant screen map excerpt.

There are several issues with this pattern. Deep, list-based navigation, known as *drill-down list navigation*, where lists lead to more lists which lead to even more lists, is often inefficient and cumbersome. The number of touches required to access a piece of content with this kind of navigation is generally very high, leading to a poor user experience—especially for users on-the-go.

Using vertical lists can also lead to awkward user interactions and poor use of whitespace on larger screens, as list items generally span the entire width of the screen yet have a fixed height. One way to alleviate this is to provide additional information, such as text summaries, that fills the available horizontal space. Another way is to provide additional information in a separate horizontal pane adjacent to the list.

Tabs

Tab Design

For design guidelines, read Android Design's Tabs guide.

Using tabs is a very popular solution for lateral navigation. This pattern allows grouping of sibling screens, in that the tab content container in the parent screen can embed child screens that otherwise would be entirely separate contexts. Tabs are most appropriate for small sets (4 or fewer) of section-related screens.

Figure 5. Example phone and tablet tab-based navigation interfaces with relevant screen map excerpt.

Several best practices apply when using tabs. Tabs should be persistent across immediate related screens. Only the designated content region should change when selecting a tab, and tab indicators should remain available at all times. Additionally, tab switches should not be treated as history. For example, if a user switches from a tab *A* to another tab *B*, pressing the *Back* button (more on that in the next lesson) should not re-select tab *A*. Tabs are usually laid out horizontally, although other presentations of tab navigation such as using a drop-down list in the Action Bar (pattern docs at Android Design) are sometimes appropriate. Lastly, and most importantly, *tabs should always run along the top of the screen*, and should not be aligned to the bottom of the screen.

There are some obvious immediate benefits of tabs over simpler list- and button-based navigation:

Since there is a single, initially-selected tab, users have immediate access to that tab's content from the parent screen.
- Users can navigate quickly between related screens, without needing to first revisit the parent.

Note: when switching tabs, it is important to maintain this tab-switching immediacy; do not block access to tab indicators by showing modal dialogs while loading content.

A common criticism is that space must be reserved for the tab indicators, detracting from the space available to tab contents. This consequence is usually acceptable, and the tradeoff commonly weighs in favor of using this pattern. You should also feel free to customize tab indicators, showing text and/or icons to make optimal use of vertical space. When adjusting indicator heights however, ensure that tab indicators are large enough for a human finger to touch without error.

Horizontal Paging (Swipe Views)

Swipe Views Design

For design guidelines, read Android Design's Swipe Views pattern guides.

Another popular lateral navigation pattern is horizontal paging, also referred to as swipe views. This pattern applies best to collection-related sibling screens, such as a list of categories (world, business, technology, and health stories). Like tabs, this pattern also allows grouping screens in that the parent presents the contents of child screens embedded within its own layout.

Figure 6. Example horizontal paging navigation interface with relevant screen map excerpt.

In a horizontal paging UI, a single child screen (referred to as a *page* here) is presented one at a time. Users are able to navigate to sibling screens by touching and dragging the screen horizontally in the direction of the desired adjacent page. This gestural interaction is often complemented by another UI element indicating the current page and available pages, to aid discoverability and provide more context to the user. This practice is especially necessary when using this pattern for lateral navigation of section-related sibling screens. Examples of such elements include tick marks, scrolling labels, and tabs.

Figure 7. Example paging companion UI elements.

It's also best to avoid this pattern when child screens contain horizontal panning surfaces (such as maps), as these conflicting interactions may

deter your screen's usability.

Additionally, for sibling-related screens, horizontal paging is most appropriate where there is some similarity in content type and when the number of siblings is relatively small. In these cases, this pattern can be used along with tabs above the content region to maximize the interface's intuitiveness. For collection-related screens, horizontal paging is most intuitive when there is a natural ordered relationship between screens, for example if each page represents consecutive calendar days. For infinite collections (again, calendar days), especially those with content in both directions, this paging mechanism can work quite well.

In the next lesson, we discuss mechanisms for allowing users to navigate up our information hierarchy and back, to previously visited screens.

Next Get started Next class

Providing Ancestral and Temporal Navigation

Previous Next Get started

This lesson teaches you to:

1. Support Temporal Navigation: *Back*
2. Provide Ancestral Navigation: *Up* and *Home*

You should also read

- Android Design: Navigation
- Tasks and Back Stack

Now that users can navigate deep into the application's screen hierarchy, we need to provide a method for navigating up the hierarchy, to parent and ancestor screens. Additionally, we should ensure that temporal navigation via the *Back* button is respected to respect Android conventions.

Back/Up Navigation Design

For design guidelines, read Android Design's Navigation pattern guide.

Support Temporal Navigation: *Back*

Temporal navigation, or navigation between historical screens, is deeply rooted in the Android system. All Android users expect the *Back* button to take them to the previous screen, regardless of other state. The set of historical screens is always rooted at the user's Launcher application (the phone's "home" screen). That is, pressing *Back* enough times should land

you back at the Launcher, after which the *Back* button will do nothing.

Figure 1. The *Back* button behavior after entering the Email app from the People (or Contacts) app.

Applications generally don't have to worry about managing the *Back* button themselves; the system handles tasks and the *back stack*, or the list of previous screens, automatically. The *Back* button by default simply traverses this list of screens, removing the current screen from the list upon being pressed.

There are, however, cases where you may want to override the behavior for *Back*. For example, if your screen contains an embedded web browser where users can interact with page elements to navigate between web pages, you may wish to trigger the embedded browser's default *back* behavior when users press the device's *Back* button. Upon reaching the beginning of the browser's internal history, you should always defer to the system's default behavior for the *Back* button.

Provide Ancestral Navigation: *Up* and *Home*

Before Android 3.0, the most common form of ancestral navigation was the *Home* metaphor. This was generally implemented as a *Home* item accessible via the device's *Menu* button, or a *Home* button at the top-left of the screen, usually as a component of the Action Bar (pattern docs at Android Design). Upon selecting *Home*, the user would be taken to the screen at the top of the screen hierarchy, generally known as the application's home screen.

Providing direct access to the application's home screen can give the user a sense of comfort and security. Regardless of where they are in the

application, if they get lost in the app, they can select *Home* to arrive back at the familiar home screen.

Android 3.0 introduced the *Up* metaphor, which is presented in the Action Bar as a substitute for the *Home* button described above. Upon tapping *Up*, the user should be taken to the parent screen in the hierarchy. This navigation step is usually the previous screen (as described with the *Back* button discussion above), but this is not universally the case. Thus, developers must ensure that *Up* for each screen navigates to a single, predetermined parent screen.

Figure 2. Example behavior for up navigation after entering the Email app from the People app.

In some cases, it's appropriate for *Up* to perform an action rather than navigating to a parent screen. Take for example, the Gmail application for Android 3.0-based tablets. When viewing a mail conversation while holding the device in landscape, the conversation list, as well as the conversation details are presented side-by-side. This is a form of parent-child screen grouping, as discussed in a previous lesson. However, when viewing a mail conversation in the portrait orientation, only the conversation details are shown. The *Up* button is used to temporarily show the parent pane, which slides in from the left of the screen. Pressing the *Up* button again while the left pane is visible exits the context of the individual conversation, up to a full-screen list of conversations.

Implementation Note: As a best practice, when implementing either *Home* or *Up*, make sure to clear the back stack of any descendent screens. For *Home*, the only remaining screen on the back stack should be the home screen. For *Up* navigation, the current screen should be removed from the back stack, unless *Back* navigates across screen hierarchies. You can use the `FLAG_ACTIVITY_CLEAR_TOP` and `FLAG_ACTIVITY_NEW_TASK` intent flags together to achieve this.

In the last lesson, we apply the concepts discussed in all of the lessons so far to create interaction design wireframes for our example news application.

Next Get started Next class

Putting it All Together: Wireframing the Example App

Previous Next Get started

This lesson teaches you to:

1. Choose Patterns
2. Sketch and Wireframe
3. Create Digital Wireframes

Now that we have a solid understanding of navigation patterns and screen grouping techniques, it's time to apply them to our screens. Let's take another look at our exhaustive screen map for the example news application from the first lesson, below.

Figure 1. Exhaustive screen map for the example news application.

Our next step is to choose and apply navigation patterns discussed in the previous lessons to this screen map, maximizing navigation speed and minimizing the number of touches to access data, while keeping the interface intuitive and consistent with Android best practices. We also need to make different choices for our different target device form factors. For simplicity, let's focus on tablets and handsets.

Choose Patterns

First, our second-level screens (*Story Category List*, *Photo List*, and *Saved Item List*) can be grouped together using tabs. Note that we don't necessarily have to use horizontally

arranged tabs; in some cases a drop-down list UI element can serve as a suitable replacement, especially on devices with narrow screens such as handsets. We can also group the *Saved Photo List* and *Saved Story List* screens together using tabs on handsets, or use multiple vertical content panes on tablets.

Finally, let's look at how we present news stories. The first option to simplify navigation across different story categories is to use horizontal paging, with a set of labels above the horizontal swiping surface, indicating the currently visible and adjacently accessible categories. On tablets in the landscape orientation, we can go a step further and present the horizontally-pageable *Story List* screen as a left pane, and the *Story View* screen as the primary content pane on the right.

Below are diagrams representing the new screen maps for handsets and tablets after applying these navigation patterns.

Figure 2. Final screen map for the example news application on handsets.

Figure 3. Final screen map for the example news application on tablets, in landscape.

At this point, it's a good idea to think of screen map variations, in case your chosen patterns don't apply well in practice (when you sketch the application's screen layouts). Below is an example screen map variation for tablets that presents story lists for different categories side-by-side, with story view screens remaining independent.

Figure 4. Example alternate screen map for tablets, in landscape.

Sketch and Wireframe

Wireframing is the step in the design process where you begin to lay out your screens. Get creative and begin imagining how to arrange UI elements to allow users to navigate your app. Keep in mind that at this point, pixel-perfect precision (creating high-fidelity mockups) is not important.

The easiest and fastest way to get started is to sketch out your screens by hand using paper and pencils. Once you begin sketching, you may uncover practicality issues in your original screen map or decisions on which patterns to use. In some cases, patterns may apply well to a given design problem in theory, but in practice they may break down and cause visual clutter or interactional issues (for example, if there are two rows of tabs on the screen). If that happens, explore other navigation patterns, or variations on chosen patterns, to arrive at a more optimal set of sketches.

After you're satisfied with initial sketches, it's a good idea to move on to digital wireframing using software such as Adobe® Illustrator, Adobe® Fireworks, OmniGraffle, or any other vector illustration tools. When choosing which tool to use, consider the following features:

- Are interactive wireframes possible? Tools such as Adobe® Fireworks offer this functionality.
- Is there screen 'master' functionality, allowing re-use of visual elements across different screens? For example, Action Bars should be visible on almost every screen in your application.
- What's the learning curve? Professional vector illustration tools may have a steep learning curve, while tools designed for wireframing may offer a smaller set of features that are more relevant to the task.

Lastly, the XML Layout Editor that comes with the Android Development Tools (ADT) plugin for Eclipse can often be used for prototyping. However, you should be careful to focus more on the high-level layout and less on visual design details at this point.

Create Digital Wireframes

After sketching out layouts on paper and choosing a digital wireframing tool that works for you, you can create the digital wireframes that will serve as the starting point for your application's visual design. Below are example wireframes for our news application, corresponding one-to-one with our screen maps from earlier in this lesson.

Figure 5. Example news application wireframes, for handsets in portrait. (**Download SVG**)

Figure 6. Example news application wireframes, for tablets in landscape. Also includes an alternate layout for presenting story lists. **(Download SVG)**

(Download SVG for device wireframe art)

Next Steps

Now that you've designed effective and intuitive intra-app navigation for your application, you can begin to spend time refining the user interface for each individual screen. For example, you can choose to use richer widgets in place of simple text labels, images, and buttons when displaying interactive content. You can also begin defining the visual styling of your application, incorporating elements from your brand's visual language in the process.

Lastly, it may be time to begin implementing your designs and writing the code for the application using the Android SDK. To get started, take a look at the following resources:

- Developer's Guide: User Interface: learn how to implement your user interface designs using the Android SDK.
- Action Bar: implement tabs, up navigation, on-screen actions, etc.
- Fragments: implement re-usable, multi-pane layouts
- Support Library: implement horizontal paging (swipe views) using `ViewPager`

Next Get started Next class

Implementing Effective Navigation

Previous Next Get started

Dependencies and prerequisites

- Android 2.2 or higher
- Understanding of fragments and Android layouts
- Android Support Library
- Designing Effective Navigation

You should also read

- Action Bar
- Fragments
- Designing for Multiple Screens

Try it out

Download the sample app

EffectiveNavigation.zip

This class demonstrates how to implement the key navigation design patterns detailed in the Designing Effective Navigation class.

After reading the lessons in this class, you should have a strong understanding of how to implement navigation patterns with tabs, swipe views, and a navigation drawer. You should also understand how to provide proper *Up* and *Back* navigation.

Note: Several elements of this class require the Support Library APIs. If you have not used the Support Library before, follow the instructions in the Support Library Setup document.

Lessons

Creating Swipe Views with Tabs
Learn how to implement tabs in the action bar and provide horizontal paging (swipe views) to navigate between tabs.

Creating a Navigation Drawer
Learn how to build an interface with a hidden navigation drawer on the side of the screen that opens with a swipe or by pressing the action bar's app icon.

Providing Up Navigation
Learn how to implement *Up* navigation using the action bar's app icon.

Providing Proper Back Navigation
Learn how to correctly handle the *Back* button in special cases, including how to insert activities into the back stack when deep-linking the user from notifications or app widgets.

Implementing Descendant Navigation
Learn the finer points of navigating down into your application's information hierarchy.

Next Get started Next class

Creating Swipe Views with Tabs

Previous Next Get started

This lesson teaches you to

1. Implement Swipe Views
2. Add Tabs to the Action Bar
3. Change Tabs with Swipe Views
4. Use a Title Strip Instead of Tabs

You should also read

- Providing Descendant and Lateral Navigation
- Android Design: Tabs
- Android Design: Swipe Views

Try it out

Download the sample app

EffectiveNavigation.zip

Swipe views provide lateral navigation between sibling screens such as tabs with a horizontal finger gesture (a pattern sometimes known as horizontal paging). This lesson teaches you how to create a tab layout with swipe views for switching between tabs, or how to show a title strip instead of tabs.

Swipe View Design

Before implementing these features, you should understand the concepts and recommendations as described in Designing Effective Navigation and

the Swipe Views design guide.

Implement Swipe Views

You can create swipe views in your app using the `ViewPager` widget, available in the Support Library. The `ViewPager` is a layout widget in which each child view is a separate page (a separate tab) in the layout.

To set up your layout with `ViewPager`, add a `<ViewPager>` element to your XML layout. For example, if each page in the swipe view should consume the entire layout, then your layout looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.view.ViewPager
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

To insert child views that represent each page, you need to hook this layout to a `PagerAdapter`. There are two kinds of adapter you can use:

`FragmentPagerAdapter`
: This is best when navigating between sibling screens representing a fixed, small number of pages.

`FragmentStatePagerAdapter`
: This is best for paging across a collection of objects for which the number of pages is undetermined. It destroys fragments as the user navigates to other pages, minimizing memory usage.

For example, here's how you might use `FragmentStatePagerAdapter` to swipe across a collection of `Fragment` objects:

```
public class CollectionDemoActivity extends FragmentActivity {
    // When requested, this adapter returns a DemoObjectFragment,
    // representing an object in the collection.
    DemoCollectionPagerAdapter mDemoCollectionPagerAdapter;
    ViewPager mViewPager;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_collection_demo);

        // ViewPager and its adapters use support library
        // fragments, so use getSupportFragmentManager.
```

```
        mDemoCollectionPagerAdapter =
                new DemoCollectionPagerAdapter(
                        getSupportFragmentManager());
        mViewPager = (ViewPager) findViewById(R.id.pager);
        mViewPager.setAdapter(mDemoCollectionPagerAdapter);
    }
}

// Since this is an object collection, use a
FragmentStatePagerAdapter,
// and NOT a FragmentPagerAdapter.
public class DemoCollectionPagerAdapter extends
FragmentStatePagerAdapter {
    public DemoCollectionPagerAdapter(FragmentManager fm) {
        super(fm);
    }

    @Override
    public Fragment getItem(int i) {
        Fragment fragment = new DemoObjectFragment();
        Bundle args = new Bundle();
        // Our object is just an integer :-P
        args.putInt(DemoObjectFragment.ARG_OBJECT, i + 1);
        fragment.setArguments(args);
        return fragment;
    }

    @Override
    public int getCount() {
        return 100;
    }

    @Override
    public CharSequence getPageTitle(int position) {
        return "OBJECT " + (position + 1);
    }
}

// Instances of this class are fragments representing a single
// object in our collection.
public static class DemoObjectFragment extends Fragment {
    public static final String ARG_OBJECT = "object";

    @Override
    public View onCreateView(LayoutInflater inflater,
            ViewGroup container, Bundle savedInstanceState) {
        // The last two arguments ensure LayoutParams are inflated
        // properly.
        View rootView = inflater.inflate(
                R.layout.fragment_collection_object, container,
false);
        Bundle args = getArguments();
        ((TextView)
rootView.findViewById(android.R.id.text1)).setText(
                Integer.toString(args.getInt(ARG_OBJECT)));
        return rootView;
    }
}
```

This example shows only the code necessary to create the swipe views.

The following sections show how you can add tabs to help facilitate navigation between pages.

Add Tabs to the Action Bar

Action bar tabs offer users a familiar interface for navigating between and identifying sibling screens in your app.

To create tabs using `ActionBar`, you need to enable `NAVIGATION_MODE_TABS`, then create several instances of `ActionBar.Tab` and supply an implementation of the `ActionBar.TabListener` interface for each one. For example, in your activity's `onCreate()` method, you can use code similar to this:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    final ActionBar actionBar = getActionBar();
    ...

    // Specify that tabs should be displayed in the action bar.
    actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);

    // Create a tab listener that is called when the user changes
tabs.
    ActionBar.TabListener tabListener = new ActionBar.TabListener()
{
        public void onTabSelected(ActionBar.Tab tab,
FragmentTransaction ft) {
            // show the given tab
        }

        public void onTabUnselected(ActionBar.Tab tab,
FragmentTransaction ft) {
            // hide the given tab
        }

        public void onTabReselected(ActionBar.Tab tab,
FragmentTransaction ft) {
            // probably ignore this event
        }
    };

    // Add 3 tabs, specifying the tab's text and TabListener
    for (int i = 0; i < 3; i++) {
        actionBar.addTab(
                actionBar.newTab()
                        .setText("Tab " + (i + 1))
                        .setTabListener(tabListener));
    }
}
```

How you handle the `ActionBar.TabListener` callbacks to change tabs depends on how you've constructed your content. But if you're using fragments for each tab with `ViewPager` as shown above, the following section shows how to switch between pages when the user selects a tab and also update the selected tab when the user swipes between pages.

Change Tabs with Swipe Views

To switch between pages in a `ViewPager` when the user selects a tab, implement your `ActionBar.TabListener` to select the appropriate page by calling `setCurrentItem()` on your `ViewPager`:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    ...

    // Create a tab listener that is called when the user changes
tabs.
    ActionBar.TabListener tabListener = new ActionBar.TabListener()
{
        public void onTabSelected(ActionBar.Tab tab,
FragmentTransaction ft) {
            // When the tab is selected, switch to the
            // corresponding page in the ViewPager.
            mViewPager.setCurrentItem(tab.getPosition());
        }
        ...
    };
}
```

Likewise, you should select the corresponding tab when the user swipes between pages with a touch gesture. You can set up this behavior by implementing the `ViewPager.OnPageChangeListener` interface to change the current tab each time the page changes. For example:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    ...

    mViewPager = (ViewPager) findViewById(R.id.pager);
    mViewPager.setOnPageChangeListener(
            new ViewPager.SimpleOnPageChangeListener() {
                @Override
                public void onPageSelected(int position) {
                    // When swiping between pages, select the
                    // corresponding tab.

getActionBar().setSelectedNavigationItem(position);
                }
            });
```

```
    ...
}
```

Use a Title Strip Instead of Tabs

If you don't want to include action bar tabs and prefer to provide scrollable tabs for a shorter visual profile, you can use `PagerTitleStrip` with your swipe views.

Below is an example layout XML file for an activity whose entire contents are a `ViewPager` and a top-aligned `PagerTitleStrip` inside it. Individual pages (provided by the adapter) occupy the remaining space inside the `ViewPager`.

```
<android.support.v4.view.ViewPager
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <android.support.v4.view.PagerTitleStrip
        android:id="@+id/pager_title_strip"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_gravity="top"
        android:background="#33b5e5"
        android:textColor="#fff"
        android:paddingTop="4dp"
        android:paddingBottom="4dp" />

</android.support.v4.view.ViewPager>
```

Next Get started Next class

Creating a Navigation Drawer

Previous Next Get started

This lesson teaches you to:

1. Create a Drawer Layout
2. Initialize the Drawer List
3. Handle Navigation Click Events
4. Listen for Open and Close Events
5. Open and Close with the App Icon

Try it out

Download the sample app

NavigationDrawer.zip

Download the Action Bar Icon Pack

Android_Design_Icons_20130926.zip

The navigation drawer is a panel that displays the app's main navigation options on the left edge of the screen. It is hidden most of the time, but is revealed when the user swipes a finger from the left edge of the screen or, while at the top level of the app, the user touches the app icon in the action bar.

This lesson describes how to implement a navigation drawer using the `DrawerLayout` APIs available in the Support Library.

Navigation Drawer Design

Before you decide to use a navigation drawer in your app, you should

understand the use cases and design principles defined in the Navigation Drawer design guide.

Create a Drawer Layout

To add a navigation drawer, declare your user interface with a `DrawerLayout` object as the root view of your layout. Inside the `DrawerLayout`, add one view that contains the main content for the screen (your primary layout when the drawer is hidden) and another view that contains the contents of the navigation drawer.

For example, the following layout uses a `DrawerLayout` with two child views: a `FrameLayout` to contain the main content (populated by a `Fragment` at runtime), and a `ListView` for the navigation drawer.

```
<android.support.v4.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <!-- The main content view -->
    <FrameLayout
        android:id="@+id/content_frame"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
    <!-- The navigation drawer -->
    <ListView android:id="@+id/left_drawer"
        android:layout_width="240dp"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        android:choiceMode="singleChoice"
        android:divider="@android:color/transparent"
        android:dividerHeight="0dp"
        android:background="#111"/>
</android.support.v4.widget.DrawerLayout>
```

This layout demonstrates some important layout characteristics:

- The main content view (the `FrameLayout` above) **must be the first child** in the `DrawerLayout` because the XML order implies z-ordering and the drawer must be on top of the content.
- The main content view is set to match the parent view's width and height, because it represents the entire UI when the navigation drawer is hidden.
- The drawer view (the `ListView`) **must specify its horizontal**

gravity with the `android:layout_gravity` attribute. To support right-to-left (RTL) languages, specify the value with `"start"` instead of `"left"` (so the drawer appears on the right when the layout is RTL).

- The drawer view specifies its width in `dp` units and the height matches the parent view. The drawer width should be no more than 320dp so the user can always see a portion of the main content.

Initialize the Drawer List

In your activity, one of the first things to do is initialize the navigation drawer's list of items. How you do so depends on the content of your app, but a navigation drawer often consists of a `ListView`, so the list should be populated by an `Adapter` (such as `ArrayAdapter` or `SimpleCursorAdapter`).

For example, here's how you can initialize the navigation list with a string array:

```
public class MainActivity extends Activity {
    private String[] mPlanetTitles;
    private DrawerLayout mDrawerLayout;
    private ListView mDrawerList;
    ...

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        mPlanetTitles =
getResources().getStringArray(R.array.planets_array);
        mDrawerLayout = (DrawerLayout)
findViewById(R.id.drawer_layout);
        mDrawerList = (ListView) findViewById(R.id.left_drawer);

        // Set the adapter for the list view
        mDrawerList.setAdapter(new ArrayAdapter<String>(this,
                R.layout.drawer_list_item, mPlanetTitles));
        // Set the list's click listener
        mDrawerList.setOnItemClickListener(new
DrawerItemClickListener());

        ...
    }
}
```

This code also calls `setOnItemClickListener()` to receive click events in

the navigation drawer's list. The next section shows how to implement this interface and change the content view when the user selects an item.

Handle Navigation Click Events

When the user selects an item in the drawer's list, the system calls `onItemClick()` on the `OnItemClickListener` given to `setOnItemClickListener()`.

What you do in the `onItemClick()` method depends on how you've implemented your app structure. In the following example, selecting each item in the list inserts a different `Fragment` into the main content view (the `FrameLayout` element identified by the `R.id.content_frame` ID):

```
private class DrawerItemClickListener implements
ListView.OnItemClickListener {
    @Override
    public void onItemClick(AdapterView parent, View view, int
position, long id) {
        selectItem(position);
    }
}

/** Swaps fragments in the main content view */
private void selectItem(int position) {
    // Create a new fragment and specify the planet to show based on
position
    Fragment fragment = new PlanetFragment();
    Bundle args = new Bundle();
    args.putInt(PlanetFragment.ARG_PLANET_NUMBER, position);
    fragment.setArguments(args);

    // Insert the fragment by replacing any existing fragment
    FragmentManager fragmentManager = getFragmentManager();
    fragmentManager.beginTransaction()
                   .replace(R.id.content_frame, fragment)
                   .commit();

    // Highlight the selected item, update the title, and close the
drawer
    mDrawerList.setItemChecked(position, true);
    setTitle(mPlanetTitles[position]);
    mDrawerLayout.closeDrawer(mDrawerList);
}

@Override
public void setTitle(CharSequence title) {
    mTitle = title;
    getActionBar().setTitle(mTitle);
}
```

Listen for Open and Close Events

To listen for drawer open and close events, call `setDrawerListener()` on your `DrawerLayout` and pass it an implementation of `DrawerLayout.DrawerListener`. This interface provides callbacks for drawer events such as `onDrawerOpened()` and `onDrawerClosed()`.

However, rather than implementing the `DrawerLayout.DrawerListener`, if your activity includes the action bar, you can instead extend the `ActionBarDrawerToggle` class. The `ActionBarDrawerToggle` implements `DrawerLayout.DrawerListener` so you can still override those callbacks, but it also facilitates the proper interaction behavior between the action bar icon and the navigation drawer (discussed further in the next section).

As discussed in the Navigation Drawer design guide, you should modify the contents of the action bar when the drawer is visible, such as to change the title and remove action items that are contextual to the main content. The following code shows how you can do so by overriding `DrawerLayout.DrawerListener` callback methods with an instance of the `ActionBarDrawerToggle` class:

```
public class MainActivity extends Activity {
    private DrawerLayout mDrawerLayout;
    private ActionBarDrawerToggle mDrawerToggle;
    private CharSequence mDrawerTitle;
    private CharSequence mTitle;
    ...

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ...

        mTitle = mDrawerTitle = getTitle();
        mDrawerLayout = (DrawerLayout)
findViewById(R.id.drawer_layout);
        mDrawerToggle = new ActionBarDrawerToggle(this,
mDrawerLayout,
                R.drawable.ic_drawer, R.string.drawer_open,
R.string.drawer_close) {

            /** Called when a drawer has settled in a completely
closed state. */
            public void onDrawerClosed(View view) {
                super.onDrawerClosed(view);
                getActionBar().setTitle(mTitle);
```

```
                invalidateOptionsMenu(); // creates call to
onPrepareOptionsMenu()
            }

            /** Called when a drawer has settled in a completely
open state. */
            public void onDrawerOpened(View drawerView) {
                super.onDrawerOpened(drawerView);
                getActionBar().setTitle(mDrawerTitle);
                invalidateOptionsMenu(); // creates call to
onPrepareOptionsMenu()
            }
        };

        // Set the drawer toggle as the DrawerListener
        mDrawerLayout.setDrawerListener(mDrawerToggle);
    }

    /* Called whenever we call invalidateOptionsMenu() */
    @Override
    public boolean onPrepareOptionsMenu(Menu menu) {
        // If the nav drawer is open, hide action items related to
the content view
        boolean drawerOpen = mDrawerLayout.isDrawerOpen(mDrawerList);
        menu.findItem(R.id.action_websearch).setVisible(!drawerOpen);
        return super.onPrepareOptionsMenu(menu);
    }
}
```

The next section describes the `ActionBarDrawerToggle` constructor arguments and the other steps required to set it up to handle interaction with the action bar icon.

Open and Close with the App Icon

Users can open and close the navigation drawer with a swipe gesture from or towards the left edge of the screen, but if you're using the action bar, you should also allow users to open and close it by touching the app icon. And the app icon should also indicate the presence of the navigation drawer with a special icon. You can implement all this behavior by using the `ActionBarDrawerToggle` shown in the previous section.

To make `ActionBarDrawerToggle` work, create an instance of it with its constructor, which requires the following arguments:

- The `Activity` hosting the drawer.
- The `DrawerLayout`.
- A drawable resource to use as the drawer indicator.

The standard navigation drawer icon is available in the Download the Action Bar Icon Pack.

- A String resource to describe the "open drawer" action (for accessibility).
- A String resource to describe the "close drawer" action (for accessibility).

Then, whether or not you've created a subclass of `ActionBarDrawerToggle` as your drawer listener, you need to call upon your `ActionBarDrawerToggle` in a few places throughout your activity lifecycle:

```
public class MainActivity extends Activity {
    private DrawerLayout mDrawerLayout;
    private ActionBarDrawerToggle mDrawerToggle;
    ...

    public void onCreate(Bundle savedInstanceState) {
        ...

        mDrawerLayout = (DrawerLayout)
findViewById(R.id.drawer_layout);
        mDrawerToggle = new ActionBarDrawerToggle(
                this,                  /* host Activity */
                mDrawerLayout,         /* DrawerLayout object */
                R.drawable.ic_drawer,  /* nav drawer icon to replace
'Up' caret */
                R.string.drawer_open,  /* "open drawer" description
*/
                R.string.drawer_close  /* "close drawer" description
*/
                ) {

            /** Called when a drawer has settled in a completely
closed state. */
            public void onDrawerClosed(View view) {
                super.onDrawerClosed(view);
                getActionBar().setTitle(mTitle);
            }

            /** Called when a drawer has settled in a completely
open state. */
            public void onDrawerOpened(View drawerView) {
                super.onDrawerOpened(drawerView);
                getActionBar().setTitle(mDrawerTitle);
            }
        };

        // Set the drawer toggle as the DrawerListener
        mDrawerLayout.setDrawerListener(mDrawerToggle);

        getActionBar().setDisplayHomeAsUpEnabled(true);
        getActionBar().setHomeButtonEnabled(true);
    }
```

```
    @Override
    protected void onPostCreate(Bundle savedInstanceState) {
        super.onPostCreate(savedInstanceState);
        // Sync the toggle state after onRestoreInstanceState has
occurred.
        mDrawerToggle.syncState();
    }

    @Override
    public void onConfigurationChanged(Configuration newConfig) {
        super.onConfigurationChanged(newConfig);
        mDrawerToggle.onConfigurationChanged(newConfig);
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Pass the event to ActionBarDrawerToggle, if it returns
        // true, then it has handled the app icon touch event
        if (mDrawerToggle.onOptionsItemSelected(item)) {
          return true;
        }
        // Handle your other action bar items...

        return super.onOptionsItemSelected(item);
    }

    ...
}
```

For a complete example of a navigation drawer, download the sample available at the top of the page.

Next Get started Next class

Providing Up Navigation

Previous Next Get started

This lesson teaches you to:

1. Specify the Parent Activity
2. Add Up Action
3. Navigate Up to Parent Activity

You should also read

- Providing Ancestral and Temporal Navigation
- Tasks and Back Stack
- Android Design: Navigation

Try it out

Download the sample app

EffectiveNavigation.zip

All screens in your app that are not the main entrance to your app (the "home" screen) should offer the user a way to navigate to the logical parent screen in the app's hierarchy by pressing the *Up* button in the action bar. This lesson shows you how to properly implement this behavior.

Up Navigation Design

The concepts and principles for *Up* navigation are described in Designing Effective Navigation and the Navigation design guide.

Figure 1. The *Up* button in the action bar.

Specify the Parent Activity

To implement *Up* navigation, the first step is to declare which activity is the appropriate parent for each activity. Doing so allows the system to facilitate navigation patterns such as *Up* because the system can determine the logical parent activity from the manifest file.

Beginning in Android 4.1 (API level 16), you can declare the logical parent of each activity by specifying the `android:parentActivityName` attribute in the `<activity>` element.

If your app supports Android 4.0 and lower, include the Support Library with your app and add a `<meta-data>` element inside the `<activity>`. Then specify the parent activity as the value for `android.support.PARENT_ACTIVITY`, matching the `android:parentActivityName` attribute.

For example:

```
<application ... >
    ...
    <!-- The main/home activity (it has no parent activity) -->
    <activity
        android:name="com.example.myfirstapp.MainActivity" ...>
        ...
    </activity>
    <!-- A child of the main activity -->
    <activity
        android:name="com.example.myfirstapp.DisplayMessageActivity"
        android:label="@string/title_activity_display_message"

android:parentActivityName="com.example.myfirstapp.MainActivity" >
        <!-- Parent activity meta-data to support 4.0 and lower -->
        <meta-data
            android:name="android.support.PARENT_ACTIVITY"
            android:value="com.example.myfirstapp.MainActivity" />
    </activity>
```

```
</application>
```

With the parent activity declared this way, you can navigate *Up* to the appropriate parent using the `NavUtils` APIs, as shown in the following sections.

Add Up Action

To allow *Up* navigation with the app icon in the action bar, call `setDisplayHomeAsUpEnabled()`:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    ...
    getActionBar().setDisplayHomeAsUpEnabled(true);
}
```

This adds a left-facing caret alongside the app icon and enables it as an action button such that when the user presses it, your activity receives a call to `onOptionsItemSelected()`. The ID for the action is `android.R.id.home`.

Navigate Up to Parent Activity

To navigate up when the user presses the app icon, you can use the `NavUtils` class's static method, `navigateUpFromSameTask()`. When you call this method, it finishes the current activity and starts (or resumes) the appropriate parent activity. If the target parent activity is in the task's back stack, it is brought forward. The way it is brought forward depends on whether the parent activity is able to handle an `onNewIntent()` call:

- If the parent activity has launch mode `<singleTop>`, or the `up` intent contains `FLAG_ACTIVITY_CLEAR_TOP`, the parent activity is brought to the top of the stack, and receives the intent through its `onNewIntent()` method.
- If the parent activity has launch mode `<standard>`, and the `up` intent does not contain `FLAG_ACTIVITY_CLEAR_TOP`, the parent activity is popped off the stack, and a new instance of that activity is created on top of the stack to receive the intent.

For example:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
    // Respond to the action bar's Up/Home button
    case android.R.id.home:
        NavUtils.navigateUpFromSameTask(this);
        return true;
    }
    return super.onOptionsItemSelected(item);
}
```

However, using `navigateUpFromSameTask()` is suitable **only when your app is the owner of the current task** (that is, the user began this task from your app). If that's not true and your activity was started in a task that belongs to a different app, then navigating *Up* should create a new task that belongs to your app, which requires that you create a new back stack.

Navigate up with a new back stack

If your activity provides any intent filters that allow other apps to start the activity, you should implement the `onOptionsItemSelected()` callback such that if the user presses the *Up* button after entering your activity from another app's task, your app starts a new task with the appropriate back stack before navigating up.

You can do so by first calling `shouldUpRecreateTask()` to check whether the current activity instance exists in a different app's task. If it returns true, then build a new task with `TaskStackBuilder`. Otherwise, you can use the `navigateUpFromSameTask()` method as shown above.

For example:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
    // Respond to the action bar's Up/Home button
    case android.R.id.home:
        Intent upIntent = NavUtils.getParentActivityIntent(this);
        if (NavUtils.shouldUpRecreateTask(this, upIntent)) {
            // This activity is NOT part of this app's task, so
create a new task
            // when navigating up, with a synthesized back stack.
            TaskStackBuilder.create(this)
```

```
                    // Add all of this activity's parents to the
back stack
                    .addNextIntentWithParentStack(upIntent)
                    // Navigate up to the closest parent
                    .startActivities();
        } else {
            // This activity is part of this app's task, so simply
            // navigate up to the logical parent activity.
            NavUtils.navigateUpTo(this, upIntent);
        }
        return true;
    }
    return super.onOptionsItemSelected(item);
}
```

Note: In order for the `addNextIntentWithParentStack()` method to work, you must declare the logical parent of each activity in your manifest file, using the `android:parentActivityName` attribute (and corresponding `<meta-data>` element) as described above.

Next Get started Next class

Providing Proper Back Navigation

Previous Next Get started

This lesson teaches you to:

1. Synthesize a new Back Stack for Deep Links
2. Implement Back Navigation for Fragments
3. Implement Back Navigation for WebViews

You should also read

- Providing Ancestral and Temporal Navigation
- Tasks and Back Stack
- Android Design: Navigation

Back navigation is how users move backward through the history of screens they previously visited. All Android devices provide a *Back* button for this type of navigation, so **your app should not add a Back button to the UI**.

In almost all situations, the system maintains a back stack of activities while the user navigates your application. This allows the system to properly navigate backward when the user presses the *Back* button. However, there are a few cases in which your app should manually specify the *Back* behavior in order to provide the best user experience.

Back Navigation Design

Before continuing with this document, you should understand the concepts and principles for *Back* navigation as described in the Navigation design guide.

Navigation patterns that require you to manually specify the *Back* behavior include:

- When the user enters a deep-level activity directly from a notification, an app widget, or the navigation drawer.
- Certain cases in which the user navigates between fragments.
- When the user navigates web pages in a `WebView`.

How to implement proper *Back* navigation in these situations is described in the following sections.

Synthesize a new Back Stack for Deep Links

Ordinarily, the system incrementally builds the back stack as the user navigates from one activity to the next. However, when the user enters your app with a *deep link* that starts the activity in its own task, it's necessary for you to synthesize a new back stack because the activity is running in a new task without any back stack at all.

For example, when a notification takes the user to an activity deep in your app hierarchy, you should add activities into your task's back stack so that pressing *Back* navigates up the app hierarchy instead of exiting the app. This pattern is described further in the Navigation design guide.

Specify parent activities in the manifest

Beginning in Android 4.1 (API level 16), you can declare the logical parent of each activity by specifying the `android:parentActivityName` attribute in the `<activity>` element. This allows the system to facilitate navigation patterns because it can determine the logical *Back* or *Up* navigation path with this information.

If your app supports Android 4.0 and lower, include the Support Library with your app and add a `<meta-data>` element inside the `<activity>`. Then specify the parent activity as the value for `android.support.PARENT_ACTIVITY`, matching the `android:parentActivityName` attribute.

For example:

```
<application ... >
    ...
    <!-- The main/home activity (it has no parent activity) -->
    <activity
        android:name="com.example.myfirstapp.MainActivity" ...>
        ...
    </activity>
    <!-- A child of the main activity -->
    <activity
        android:name="com.example.myfirstapp.DisplayMessageActivity"
        android:label="@string/title_activity_display_message"

android:parentActivityName="com.example.myfirstapp.MainActivity" >
        <!-- The meta-data element is needed for versions lower than
4.1 -->
        <meta-data
            android:name="android.support.PARENT_ACTIVITY"
            android:value="com.example.myfirstapp.MainActivity" />
    </activity>
</application>
```

With the parent activity declared this way, you can use the `NavUtils` APIs to synthesize a new back stack by identifying which activity is the appropriate parent for each activity.

Create back stack when starting the activity

Adding activities to the back stack begins upon the event that takes the user into your app. That is, instead of calling `startActivity()`, use the `TaskStackBuilder` APIs to define each activity that should be placed into a new back stack. Then begin the target activity by calling `startActivities()`, or create the appropriate `PendingIntent` by calling `getPendingIntent()`.

For example, when a notification takes the user to an activity deep in your app hierarchy, you can use this code to create a `PendingIntent` that starts an activity and inserts a new back stack into the target task:

```
// Intent for the activity to open when user selects the
notification
Intent detailsIntent = new Intent(this, DetailsActivity.class);

// Use TaskStackBuilder to build the back stack and get the
PendingIntent
PendingIntent pendingIntent =
        TaskStackBuilder.create(this)
                        // add all of DetailsActivity's parents to
```

```
the stack,
                        // followed by DetailsActivity itself
                        .addNextIntentWithParentStack(upIntent)
                        .getPendingIntent(0,
PendingIntent.FLAG_UPDATE_CURRENT);

NotificationCompat.Builder builder = new
NotificationCompat.Builder(this);
builder.setContentIntent(pendingIntent);
...
```

The resulting `PendingIntent` specifies not only the activity to start (as defined by `detailsIntent`), but also the back stack that should be inserted into the task (all parents of the `DetailsActivity` defined by `detailsIntent`). So when the `DetailsActivity` starts, pressing *Back* navigates backward through each of the `DetailsActivity` class's parent activities.

Note: In order for the `addNextIntentWithParentStack()` method to work, you must declare the logical parent of each activity in your manifest file, using the `android:parentActivityName` attribute (and corresponding `<meta-data>` element) as described above.

Implement Back Navigation for Fragments

When using fragments in your app, individual `FragmentTransaction` objects may represent context changes that should be added to the back stack. For example, if you are implementing a master/detail flow on a handset by swapping out fragments, you should ensure that pressing the *Back* button on a detail screen returns the user to the master screen. To do so, call `addToBackStack()` before you commit the transaction:

```
// Works with either the framework FragmentManager or the
// support package FragmentManager (getSupportFragmentManager).
getSupportFragmentManager().beginTransaction()
                            .add(detailFragment, "detail")
                            // Add this transaction to the back stack
                            .addToBackStack()
                            .commit();
```

When there are `FragmentTransaction` objects on the back stack and the user presses the *Back* button, the `FragmentManager` pops the most recent transaction off the back stack and performs the reverse action (such as removing a fragment if the transaction added it).

Note: You **should not add transactions to the back stack** when the transaction is for horizontal navigation (such as when switching tabs) or when modifying the content appearance (such as when adjusting filters). For more information, about when *Back* navigation is appropriate, see the Navigation design guide.

If your application updates other user interface elements to reflect the current state of your fragments, such as the action bar, remember to update the UI when you commit the transaction. You should update your user interface after the back stack changes in addition to when you commit the transaction. You can listen for when a `FragmentTransaction` is reverted by setting up an `FragmentManager.OnBackStackChangedListener`:

```
getSupportFragmentManager().addOnBackStackChangedListener(
        new FragmentManager.OnBackStackChangedListener() {
            public void onBackStackChanged() {
                // Update your UI here.
            }
        });
```

Implement Back Navigation for WebViews

If a part of your application is contained in a `WebView`, it may be appropriate for *Back* to traverse browser history. To do so, you can override `onBackPressed()` and proxy to the `WebView` if it has history state:

```
@Override
public void onBackPressed() {
    if (mWebView.canGoBack()) {
        mWebView.goBack();
        return;
    }

    // Otherwise defer to system default behavior.
    super.onBackPressed();
}
```

Be careful when using this mechanism with highly dynamic web pages that can grow a large history. Pages that generate an extensive history, such as those that make frequent changes to the document hash, may make it tedious for users to get out of your activity.

For more information about using `WebView`, read Building Web Apps in

WebView.

Next Get started Next class

Implementing Descendant Navigation

Previous Next Get started

This lesson teaches you to:

1. Implement Master/Detail Flows Across Handsets and Tablets
2. Navigate into External Activities

You should also read

- Providing Descendant and Lateral Navigation
- Android Design: App Structure
- Android Design: Multi-pane Layouts

Try it out

Download the sample app

EffectiveNavigation.zip

Descendant navigation is navigation down the application's information hierarchy. This is described in Designing Effective Navigation and also covered in Android Design: Application Structure.

Descendant navigation is usually implemented using `Intent` objects and `startActivity()`, or by adding fragments to an activity using `FragmentTransaction` objects. This lesson covers other interesting cases that arise when implementing descendant navigation.

Implement Master/Detail Flows Across

Handsets and Tablets

In a *master/detail* navigation flow, a *master* screen contains a list of items in a collection, and a *detail* screen shows detailed information about a specific item within that collection. Implementing navigation from the master screen to the detail screen is one form of descendant navigation.

Handset touchscreens are most suitable for displaying one screen at a time (either the master or the detail screen); this concern is further discussed in Planning for Multiple Touchscreen Sizes. Descendant navigation in this case is often implemented using an `Intent` that starts an activity representing the detail screen. On the other hand, tablet displays, especially when viewed in the landscape orientation, are best suited for showing multiple content panes at a time: the master on the left, and the detail to the right). Here, descendant navigation is usually implemented using a `FragmentTransaction` that adds, removes, or replaces the detail pane with new content.

The basics of implementing this pattern are described in the Implementing Adaptive UI Flows lesson of the *Designing for Multiple Screens* class. The class describes how to implement a master/detail flow using two activities on a handset and a single activity on a tablet.

Navigate into External Activities

There are cases where descending into your application's information hierarchy leads to activities from other applications. For example, when viewing the contact details screen for an entry in the phone address book, a child screen detailing recent posts by the contact on a social network may belong to a social networking application.

When launching another application's activity to allow the user to say, compose an email or pick a photo attachment, you generally don't want the user to return to this activity if they relaunch your application from the Launcher (the device home screen). It would be confusing if touching your application icon brought the user to a "compose email" screen.

To prevent this from occurring, simply add the FLAG_ACTIVITY_CLEAR_WHEN_TASK_RESET flag to the intent used to launch the external activity, like so:

```
Intent externalActivityIntent = new Intent(Intent.ACTION_PICK);
externalActivityIntent.setType("image/*");
externalActivityIntent.addFlags(
        Intent.FLAG_ACTIVITY_CLEAR_WHEN_TASK_RESET);
startActivity(externalActivityIntent);
```

Next Get started Next class

Notifying the User

Previous Next Get started

Dependencies and prerequisites

- Android 1.6 (API Level 4) or higher

You should also read

- Notifications API Guide
- Intents and Intent Filters
- Notifications Design Guide

Try it out

Download the sample

NotifyUser.zip

A notification is a user interface element that you display outside your app's normal UI to indicate that an event has occurred. Users can choose to view the notification while using other apps and respond to it when it's convenient for them.

The Notifications design guide shows you how to design effective notifications and when to use them. This class shows you how to implement the most common notification designs.

Lessons

Building a Notification

Learn how to create a notification `Builder`, set the required features,

and issue the notification.

Preserving Navigation when Starting an Activity

Learn how to enforce the proper navigation for an `Activity` started from a notification.

Updating Notifications

Learn how to update and remove notifications.

Using Big View Styles

Learn how to create a big view within an expanded notification, while still maintaining backward compatibility.

Displaying Progress in a Notification

Learn how to display the progress of an operation in a notification, both for operations where you can estimate how much has been completed (determinate progress) and operations where you don't know how much has been completed (indefinite progress).

Next Get started Next class

Building a Notification

Previous Next Get started

This lesson teaches you to

1. Create a Notification Builder
2. Define the Notification's Action
3. Set the Notification's Click Behavior
4. Issue the Notification

You should also read

- Notifications API Guide
- Intents and Intent Filters
- Notifications Design Guide

This lesson explains how to create and issue a notification.

The examples in this class are based on the `NotificationCompat.Builder` class. `NotificationCompat.Builder` is in the Support Library. You should use `NotificationCompat` and its subclasses, particularly `NotificationCompat.Builder`, to provide the best notification support for a wide range of platforms.

Create a Notification Builder

When creating a notification, specify the UI content and actions with a `NotificationCompat.Builder` object. At bare minimum, a `Builder` object must include the following:

- A small icon, set by `setSmallIcon()`
- A title, set by `setContentTitle()`

- Detail text, set by `setContentText()`

For example:

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

Define the Notification's Action

Although actions are optional, you should add at least one action to your notification. An action takes users directly from the notification to an `Activity` in your application, where they can look at the event that caused the notification or do further work. Inside a notification, the action itself is defined by a `PendingIntent` containing an `Intent` that starts an `Activity` in your application.

How you construct the `PendingIntent` depends on what type of `Activity` you're starting. When you start an `Activity` from a notification, you must preserve the user's expected navigation experience. In the snippet below, clicking the notification opens a new activity that effectively extends the behavior of the notification. In this case there is no need to create an artificial back stack (see Preserving Navigation when Starting an Activity for more information):

```
Intent resultIntent = new Intent(this, ResultActivity.class);
...
// Because clicking the notification opens a new ("special")
activity, there's
// no need to create an artificial back stack.
PendingIntent resultPendingIntent =
    PendingIntent.getActivity(
    this,
    0,
    resultIntent,
    PendingIntent.FLAG_UPDATE_CURRENT
);
```

Set the Notification's Click Behavior

To associate the `PendingIntent` created in the previous step with a gesture, call the appropriate method of `NotificationCompat.Builder`. For

example, to start an activity when the user clicks the notification text in the notification drawer, add the `PendingIntent` by calling `setContentIntent()`. For example:

```
PendingIntent resultPendingIntent;
...
mBuilder.setContentIntent(resultPendingIntent);
```

Issue the Notification

To issue the notification:

- Get an instance of `NotificationManager`.
- Use the `notify()` method to issue the notification. When you call `notify()`, specify a notification ID. You can use this ID to update the notification later on. This is described in more detail in Managing Notifications.
- Call `build()`, which returns a `Notification` object containing your specifications.

  For example:

  ```
  NotificationCompat.Builder mBuilder;
  ...
  // Sets an ID for the notification
  int mNotificationId = 001;
  // Gets an instance of the NotificationManager service
  NotificationManager mNotifyMgr =
          (NotificationManager)
  getSystemService(NOTIFICATION_SERVICE);
  // Builds the notification and issues it.
  mNotifyMgr.notify(mNotificationId, mBuilder.build());
  ```

Next Get started Next class

Preserving Navigation when Starting an Activity

Previous Next Get started

This lesson teaches you to

1. Set up a regular activity PendingIntent
2. Set up a special activity PendingIntent

You should also read

- Notifications API Guide
- Intents and Intent Filters
- Notifications Design Guide

Part of designing a notification is preserving the user's expected navigation experience. For a detailed discussion of this topic, see the Notifications API guide. There are two general situations:

Regular activity
: You're starting an `Activity` that's part of the application's normal workflow.

Special activity
: The user only sees this `Activity` if it's started from a notification. In a sense, the `Activity` extends the notification by providing information that would be hard to display in the notification itself.

Set Up a Regular Activity PendingIntent

To set up a `PendingIntent` that starts a direct entry `Activity`, follow these steps:

1. Define your application's Activity hierarchy in the manifest. The final XML should look like this:

```
<activity
    android:name=".MainActivity"
    android:label="@string/app_name" >
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category
android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
<activity
    android:name=".ResultActivity"
    android:parentActivityName=".MainActivity">
    <meta-data
        android:name="android.support.PARENT_ACTIVITY"
        android:value=".MainActivity"/>
</activity>
```

2. Create a back stack based on the Intent that starts the Activity. For example:

```
int id = 1;
...
Intent resultIntent = new Intent(this, ResultActivity.class);
TaskStackBuilder stackBuilder = TaskStackBuilder.create(this);
// Adds the back stack
stackBuilder.addParentStack(ResultActivity.class);
// Adds the Intent to the top of the stack
stackBuilder.addNextIntent(resultIntent);
// Gets a PendingIntent containing the entire back stack
PendingIntent resultPendingIntent =
        stackBuilder.getPendingIntent(0,
PendingIntent.FLAG_UPDATE_CURRENT);
...
NotificationCompat.Builder builder = new
NotificationCompat.Builder(this);
builder.setContentIntent(resultPendingIntent);
NotificationManager mNotificationManager =
    (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);
mNotificationManager.notify(id, builder.build());
```

Set Up a Special Activity PendingIntent

A special Activity doesn't need a back stack, so you don't have to define its Activity hierarchy in the manifest, and you don't have to call addParentStack() to build a back stack. Instead, use the manifest to set up the Activity task options, and create the PendingIntent by calling getActivity():

1. In your manifest, add the following attributes to the `<activity>` element for the `Activity`:

   `android:name`="*activityclass*"
   : The activity's fully-qualified class name.

   `android:taskAffinity`=""
   : Combined with the `FLAG_ACTIVITY_NEW_TASK` flag that you set in code, this ensures that this `Activity` doesn't go into the application's default task. Any existing tasks that have the application's default affinity are not affected.

   `android:excludeFromRecents`="true"
   : Excludes the new task from *Recents*, so that the user can't accidentally navigate back to it.

   This snippet shows the element:

   ```
   <activity
       android:name=".ResultActivity"
   ...
       android:launchMode="singleTask"
       android:taskAffinity=""
       android:excludeFromRecents="true">
   </activity>
   ...
   ```

2. Build and issue the notification:
   a. Create an `Intent` that starts the `Activity`.
   b. Set the `Activity` to start in a new, empty task by calling `setFlags()` with the flags `FLAG_ACTIVITY_NEW_TASK` and `FLAG_ACTIVITY_CLEAR_TASK`.
   c. Set any other options you need for the `Intent`.
   d. Create a `PendingIntent` from the `Intent` by calling `getActivity()`. You can then use this `PendingIntent` as the argument to `setContentIntent()`.

   The following code snippet demonstrates the process:

   ```
   // Instantiate a Builder object.
   NotificationCompat.Builder builder = new
   NotificationCompat.Builder(this);
   // Creates an Intent for the Activity
   Intent notifyIntent =
           new Intent(new ComponentName(this,
   ResultActivity.class));
   ```

```
// Sets the Activity to start in a new, empty task
notifyIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK |
        Intent.FLAG_ACTIVITY_CLEAR_TASK);
// Creates the PendingIntent
PendingIntent notifyIntent =
        PendingIntent.getActivity(
        this,
        0,
        notifyIntent,
        PendingIntent.FLAG_UPDATE_CURRENT
);

// Puts the PendingIntent into the notification builder
builder.setContentIntent(notifyIntent);
// Notifications are issued by sending them to the
// NotificationManager system service.
NotificationManager mNotificationManager =
    (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);
// Builds an anonymous Notification object from the
builder, and
// passes it to the NotificationManager
mNotificationManager.notify(id, builder.build());
```

Next Get started Next class

Updating Notifications

Previous Next Get started

This lesson teaches you to

1. Modify a Notification
2. Remove Notifications

You should also read

- Notifications API Guide
- Intents and Intent Filters
- Notifications Design Guide

When you need to issue a notification multiple times for the same type of event, you should avoid making a completely new notification. Instead, you should consider updating a previous notification, either by changing some of its values or by adding to it, or both.

The following section describes how to update notifications and also how to remove them.

Modify a Notification

To set up a notification so it can be updated, issue it with a notification ID by calling `NotificationManager.notify(ID, notification)`. To update this notification once you've issued it, update or create a `NotificationCompat.Builder` object, build a `Notification` object from it, and issue the `Notification` with the same ID you used previously.

The following snippet demonstrates a notification that is updated to reflect the number of events that have occurred. It stacks the

notification, showing a summary:

```
mNotificationManager =
        (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);
// Sets an ID for the notification, so it can be updated
int notifyID = 1;
mNotifyBuilder = new NotificationCompat.Builder(this)
    .setContentTitle("New Message")
    .setContentText("You've received new messages.")
    .setSmallIcon(R.drawable.ic_notify_status)
numMessages = 0;
// Start of a loop that processes data and then notifies the user
...
    mNotifyBuilder.setContentText(currentText)
        .setNumber(++numMessages);
    // Because the ID remains unchanged, the existing notification
is
    // updated.
    mNotificationManager.notify(
            notifyID,
            mNotifyBuilder.build());
...
```

Remove Notifications

Notifications remain visible until one of the following happens:

- The user dismisses the notification either individually or by using "Clear All" (if the notification can be cleared).
- The user touches the notification, and you called `setAutoCancel()` when you created the notification.
- You call `cancel()` for a specific notification ID. This method also deletes ongoing notifications.
- You call `cancelAll()`, which removes all of the notifications you previously issued.

Next Get started Next class

Using Big View Styles

Previous Next Get started

This lesson teaches you to

1. Set Up the Notification to Launch a New Activity
2. Construct the Big View

You should also read

- Notifications API Guide
- Intents and Intent Filters
- Notifications Design Guide

Notifications in the notification drawer appear in two main visual styles, normal view and big view. The big view of a notification only appears when the notification is expanded. This happens when the notification is at the top of the drawer, or the user clicks the notification.

Big views were introduced in Android 4.1, and they're not supported on older devices. This lesson describes how to incorporate big view notifications into your app while still providing full functionality via the normal view. See the Notifications API guide for more discussion of big views.

Here is an example of a normal view:

Figure 1. Normal view notification.

Here is an example of a big view:

Figure 2. Big view notification.

In the sample application shown in this lesson, both the normal view and the big view give users access to same functionality:

- The ability to snooze or dismiss the notification.
- A way to view the reminder text the user set as part of the timer.

The normal view provides these features through a new activity that launches when the user clicks the notification. Keep this in mind as you design your notifications—first provide the functionality in the normal view, since this is how many users will interact with the notification.

Set Up the Notification to Launch a New Activity

The sample application uses an `IntentService` subclass (`PingService`) to construct and issue the notification.

In this snippet, the `IntentService` method `onHandleIntent()` specifies the new activity that will be launched if the user clicks the notification itself. The method `setContentIntent()` defines a pending intent that should be fired when the user clicks the notification, thereby launching the activity.

```
Intent resultIntent = new Intent(this, ResultActivity.class);
resultIntent.putExtra(CommonConstants.EXTRA_MESSAGE, msg);
resultIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK |
        Intent.FLAG_ACTIVITY_CLEAR_TASK);

// Because clicking the notification launches a new ("special")
activity,
```

```
// there's no need to create an artificial back stack.
PendingIntent resultPendingIntent =
         PendingIntent.getActivity(
         this,
         0,
         resultIntent,
         PendingIntent.FLAG_UPDATE_CURRENT
);

// This sets the pending intent that should be fired when the user
clicks the
// notification. Clicking the notification launches a new activity.
builder.setContentIntent(resultPendingIntent);
```

Construct the Big View

This snippet shows how to set up the buttons that will appear in the big view:

```
// Sets up the Snooze and Dismiss action buttons that will appear in
the
// big view of the notification.
Intent dismissIntent = new Intent(this, PingService.class);
dismissIntent.setAction(CommonConstants.ACTION_DISMISS);
PendingIntent piDismiss = PendingIntent.getService(this, 0,
dismissIntent, 0);

Intent snoozeIntent = new Intent(this, PingService.class);
snoozeIntent.setAction(CommonConstants.ACTION_SNOOZE);
PendingIntent piSnooze = PendingIntent.getService(this, 0,
snoozeIntent, 0);
```

This snippet shows how to construct the `Builder` object. It sets the style for the big view to be "big text," and sets its content to be the reminder message. It uses `addAction()` to add the **Snooze** and **Dismiss** buttons (and their associated pending intents) that will appear in the notification's big view:

```
// Constructs the Builder object.
NotificationCompat.Builder builder =
        new NotificationCompat.Builder(this)
        .setSmallIcon(R.drawable.ic_stat_notification)
        .setContentTitle(getString(R.string.notification))
        .setContentText(getString(R.string.ping))
        .setDefaults(Notification.DEFAULT_ALL) // requires VIBRATE
permission
        /*
         * Sets the big view "big text" style and supplies the
         * text (the user's reminder message) that will be displayed
         * in the detail area of the expanded notification.
         * These calls are ignored by the support library for
         * pre-4.1 devices.
         */
```

```
        .setStyle(new NotificationCompat.BigTextStyle()
                .bigText(msg))
        .addAction (R.drawable.ic_stat_dismiss,
                getString(R.string.dismiss), piDismiss)
        .addAction (R.drawable.ic_stat_snooze,
                getString(R.string.snooze), piSnooze);
```

Next Get started Next class

Displaying Progress in a Notification

Previous Next Get started

This lesson teaches you to

1. Display a Fixed-duration progress Indicator
2. Display a Continuing Activity Indicator

You should also read

- Notifications API Guide
- Intents and Intent Filters
- Notifications Design Guide

Notifications can include an animated progress indicator that shows users the status of an ongoing operation. If you can estimate how long the operation takes and how much of it is complete at any time, use the "determinate" form of the indicator (a progress bar). If you can't estimate the length of the operation, use the "indeterminate" form of the indicator (an activity indicator).

Progress indicators are displayed with the platform's implementation of the `ProgressBar` class.

To use a progress indicator, call `setProgress()`. The determinate and indeterminate forms are described in the following sections.

Display a Fixed-duration Progress Indicator

To display a determinate progress bar, add the bar to your notification by

calling `setProgress(max, progress, false)` and then issue the notification. The third argument is a boolean that indicates whether the progress bar is indeterminate (**true**) or determinate (**false**). As your operation proceeds, increment `progress`, and update the notification. At the end of the operation, `progress` should equal `max`. A common way to call `setProgress()` is to set `max` to 100 and then increment `progress` as a "percent complete" value for the operation.

You can either leave the progress bar showing when the operation is done, or remove it. In either case, remember to update the notification text to show that the operation is complete. To remove the progress bar, call `setProgress(0, 0, false)`. For example:

```
int id = 1;
...
mNotifyManager =
        (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);
mBuilder = new NotificationCompat.Builder(this);
mBuilder.setContentTitle("Picture Download")
    .setContentText("Download in progress")
    .setSmallIcon(R.drawable.ic_notification);
// Start a lengthy operation in a background thread
new Thread(
    new Runnable() {
        @Override
        public void run() {
            int incr;
            // Do the "lengthy" operation 20 times
            for (incr = 0; incr <= 100; incr+=5) {
                    // Sets the progress indicator to a max value,
the
                    // current completion percentage, and
"determinate"
                    // state
                    mBuilder.setProgress(100, incr, false);
                    // Displays the progress bar for the first time.
                    mNotifyManager.notify(id, mBuilder.build());
                        // Sleeps the thread, simulating an
operation
                        // that takes time
                        try {
                            // Sleep for 5 seconds
                            Thread.sleep(5*1000);
                        } catch (InterruptedException e) {
                            Log.d(TAG, "sleep failure");
                        }
            }
            // When the loop is finished, updates the notification
            mBuilder.setContentText("Download complete")
            // Removes the progress bar
                    .setProgress(0,0,false);
            mNotifyManager.notify(id, mBuilder.build());
```

```
        }
    }
// Starts the thread by calling the run() method in its Runnable
).start();
```

The resulting notifications are shown in figure 1. On the left side is a snapshot of the notification during the operation; on the right side is a snapshot of it after the operation has finished.

Figure 1. The progress bar during and after the operation.

Display a Continuing Activity Indicator

To display a continuing (indeterminate) activity indicator, add it to your notification with `setProgress(0, 0, true)` and issue the notification. The first two arguments are ignored, and the third argument declares that the indicator is indeterminate. The result is an indicator that has the same style as a progress bar, except that its animation is ongoing.

Issue the notification at the beginning of the operation. The animation will run until you modify your notification. When the operation is done, call `setProgress(0, 0, false)` and then update the notification to remove the activity indicator. Always do this; otherwise, the animation will run even when the operation is complete. Also remember to change the notification text to indicate that the operation is complete.

To see how continuing activity indicators work, refer to the preceding snippet. Locate the following lines:

```
// Sets the progress indicator to a max value, the current
completion
// percentage, and "determinate" state
mBuilder.setProgress(100, incr, false);
// Issues the notification
mNotifyManager.notify(id, mBuilder.build());
```

Replace the lines you've found with the following lines. Notice that the

third parameter in the `setProgress()` call is set to `true` to indicate that the progress bar is indeterminate:

```
 // Sets an activity indicator for an operation of indeterminate
length
mBuilder.setProgress(0, 0, true);
// Issues the notification
mNotifyManager.notify(id, mBuilder.build());
```

The resulting indicator is shown in figure 2:

Figure 2. An ongoing activity indicator.

Next Get started Next class

Supporting Swipe-to-Refresh

Previous Next Get started

Dependencies and prerequisites

- Android 1.6 (API level 4) or later
- Latest version of the Android v4 Support Library

Sample Apps

- SwipeRefreshLayoutBasic
- SwipeRefreshListFragment
- SwipeRefreshMultipleViews

Even if your app automatically updates its content on a regular basis, you can allow users to request manual updates as well. For example, a weather forecasting app can allow users get the latest forecasts on demand. To provide a standard user experience for requesting updates, the Android platform includes the swipe-to-refresh design pattern, which allows users to trigger an update with a vertical swipe.

Note: This class requires the latest version of the Android v4 Support Library APIs. If you have not used the Support Library before, follow the instructions in the Support Library Setup document.

Lessons

Adding Swipe-to-Refresh To Your App

To replay the movie, click on the device screen

Learn how to provide swipe-to-refresh support in a `ListView` or `GridView`, and how to provide an accessible refresh option using the action bar.

Responding to a Refresh Request

Learn how to respond to the swipe-to-refresh gesture, and how to perform the same update from an action bar action.

Next Get started Next class

Adding Swipe-to-Refresh To Your App

Previous Next Get started

This lesson teaches you to

1. Add the SwipeRefreshLayout Widget
2. Add a Refresh Action to the Action Bar

You should also read

- Accessibility
- Adding the Action Bar

Sample Apps

- SwipeRefreshLayoutBasic
- SwipeRefreshListFragment

The swipe-to-refresh user interface pattern is implemented entirely within the `SwipeRefreshLayout` widget, which detects the vertical swipe, displays a distinctive progress bar, and triggers callback methods in your app. You enable this behavior by adding the widget to your layout file as the parent of a `ListView` or `GridView`, and implementing the refresh behavior that gets invoked when the user swipes.

This lesson shows you how to add the widget to an existing layout. It also shows you how to add a refresh action to the action bar overflow area, so that users who may be unable to use the swipe gesture can trigger a manual update with an external device.

Add the SwipeRefreshLayout Widget

To add the swipe to refresh widget to an existing app, add `SwipeRefreshLayout` as the parent of a single `ListView` or `GridView`. Remember that `SwipeRefreshLayout` only supports a single `ListView` or `GridView` child.

The following example demonstrates how to add the `SwipeRefreshLayout` widget to an existing layout file containing a `ListView`:

```
<android.support.v4.widget.SwipeRefreshLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/swiperefresh"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <ListView
        android:id="@android:id/list"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</android.support.v4.widget.SwipeRefreshLayout>
```

You can also use the `SwipeRefreshLayout` widget with a `ListFragment`. If the layout contains a `ListView` with the ID `"@android:id/list"`, the swipe-to-refresh functionality is automatically supported. However, explicitly declaring the `ListView` in this way supersedes the default `ListFragment` view structure. If you want to use the default view structure, you will have to override parts of the `SwipeRefreshLayout` and `ListFragment` behavior. For an example of how to do this, see the SwipeRefreshListFragment sample app.

Add a Refresh Action to the Action Bar

You should add a refresh action to your app's action bar to ensure that users who may not be able to perform a swipe gesture can still trigger a manual update. For example, users with accessibility issues can trigger action bar actions using external devices, such as keyboards and D-pads.

You should add the refresh action as a menu item, rather than as a button, by setting the attribute `android:showAsAction=never`. If you display the action as a button, users may assume that the refresh button

action is different from the swipe-to-refresh action. By making the refresh action less conspicuous in the action bar, you can encourage users to perform manual updates with the swipe gesture while still maintaining the accessible option in a place where D-pad users would look for it.

The following code demonstrates how to add the swipe-to-refresh action to the overflow area:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
    <item
        android:id="@+id/menu_refresh"
        android:showAsAction="never"
        android:title="@string/menu_refresh"/>
</menu>
```

Next Get started Next class

Responding to a Refresh Request

Previous Next Get started

This lesson teaches you to

1. Respond to the Refresh Gesture
2. Respond to the Refresh Action

Sample App

- SwipeRefreshLayoutBasic

This lesson shows you how to update your app when the user requests a manual refresh, whether the user triggers the refresh with a swipe gesture or by using the action bar refresh action.

Respond to the Refresh Gesture

When the user makes a swipe gesture, the system displays the progress indicator and calls your app's callback method. Your callback method is responsible for actually updating the app's data.

To respond to the refresh gesture in your app, implement the `SwipeRefreshLayout.OnRefreshListener` interface and its `onRefresh()` method. The `onRefresh()` method is invoked when the user performs a swipe gesture.

You should put the code for the actual update operation in a separate method, and call that update method from your `onRefresh()` implementation. That way, you can use the same update method to perform the update when the user triggers a refresh from the action bar.

Your update method calls `setRefreshing(false)` when it has finished updating the data. Calling this method instructs the `SwipeRefreshLayout` to remove the progress indicator and update the view contents.

For example, the following code implements `onRefresh()` and invokes the method `myUpdateOperation()` to update the data displayed by the `ListView`:

```
/*
 * Sets up a SwipeRefreshLayout.OnRefreshListener that is invoked
when the user
 * performs a swipe-to-refresh gesture.
 */
mySwipeRefreshLayout.setOnRefreshListener(
    new SwipeRefreshLayout.OnRefreshListener() {
        @Override
        public void onRefresh() {
            Log.i(LOG_TAG, "onRefresh called from
SwipeRefreshLayout");

            // This method performs the actual data-refresh
operation.
            // The method calls setRefreshing(false) when it's
finished.
            myUpdateOperation();
        }
    }
);
```

Respond to the Refresh Action

If the user requests a refresh by using the action bar, the system calls the `onOptionsItemSelected()` method. Your app should respond to this call by displaying the progress indicator and refreshing the app's data.

To respond to the refresh action, override `onOptionsItemSelected()`. In your override method, trigger the `SwipeRefreshLayout` progress indicator by calling `setRefreshing()` with the value `true`, then perform the update operation. Once again, you should be doing the actual update in a separate method, so the same method can be called whether the user triggers the update with a swipe or by using the action bar. When the update has finished, call `setRefreshing(false)` to remove the refresh progress indicator.

The following code shows how to respond to the request action:

```
/*
 * Listen for option item selections so that we receive a
notification
 * when the user requests a refresh by selecting the refresh action
bar item.
 */
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {

        // Check if user triggered a refresh:
        case R.id.menu_refresh:
            Log.i(LOG_TAG, "Refresh menu item selected");

            // Signal SwipeRefreshLayout to start the progress
indicator
            mySwipeRefreshLayout.setRefreshing(true);

            // Start the refresh background task.
            // This method calls setRefreshing(false) when it's
finished.
            myUpdateOperation();

            return true;
    }

    // User didn't trigger a refresh, let the superclass handle this
action
    return super.onOptionsItemSelected(item);

}
```

Note: When the user triggers a refresh with a swipe action as described in Respond to the Refresh Gesture, you do not need to call `setRefreshing()`. The `SwipeRefreshLayout` widget takes care of displaying the progress indicator and removing it when the update has finished. However, if the update is triggered by any means *other than* a swipe gesture, you need to explicitly turn the progress indicator on with `setRefreshing()`. The method which actually refreshes the data calls `setRefreshing(false)` to signal that the update is finished.

Next Get started Next class

Adding Search Functionality

Previous Next Get started

Dependencies and prerequisites

- Android 3.0 or later (with some support for Android 2.1)
- Experience building an Android User Interface

You should also read

- Search
- Searchable Dictionary Sample App

Android's built-in search features offer apps an easy way to provide a consistent search experience for all users. There are two ways to implement search in your app depending on the version of Android that is running on the device. This class covers how to add search with `SearchView`, which was introduced in Android 3.0, while maintaining backward compatibility with older versions of Android by using the default search dialog provided by the system.

Lessons

Setting Up the Search Interface
: Learn how to add a search interface to your app and how to configure an activity to handle search queries.

Storing and Searching for Data
: Learn a simple way to store and search for data in a SQLite virtual database table.

Remaining Backward Compatible
: Learn how to keep search features backward compatible with older devices.

Next Get started Next class

Setting Up the Search Interface

Previous Next Get started

This lesson teaches you to

- Add the Search View to the Action Bar
- Create a Searchable Configuration
- Create a Searchable Activity

You should also read:

- Action Bar

Beginning in Android 3.0, using the `SearchView` widget as an item in the action bar is the preferred way to provide search in your app. Like with all items in the action bar, you can define the `SearchView` to show at all times, only when there is room, or as a collapsible action, which displays the `SearchView` as an icon initially, then takes up the entire action bar as a search field when the user clicks the icon.

Note: Later in this class, you will learn how to make your app compatible down to Android 2.1 (API level 7) for devices that do not support `SearchView`.

Add the Search View to the Action Bar

To add a `SearchView` widget to the action bar, create a file named `res/menu/options_menu.xml` in your project and add the following code to the file. This code defines how to create the search item, such as the icon to use and the title of the item. The `collapseActionView` attribute allows your `SearchView` to expand to take up the whole action bar and collapse back down into a normal action bar item when not in use.

Because of the limited action bar space on handset devices, using the `collapsibleActionView` attribute is recommended to provide a better user experience.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/search"
          android:title="@string/search_title"
          android:icon="@drawable/ic_search"
          android:showAsAction="collapseActionView|ifRoom"
          android:actionViewClass="android.widget.SearchView" />
</menu>
```

Note: If you already have an existing XML file for your menu items, you can add the `<item>` element to that file instead.

To display the `SearchView` in the action bar, inflate the XML menu resource (`res/menu/options_menu.xml`) in the `onCreateOptionsMenu()` method of your activity:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.options_menu, menu);

    return true;
}
```

If you run your app now, the `SearchView` appears in your app's action bar, but it isn't functional. You now need to define *how* the `SearchView` behaves.

Create a Searchable Configuration

A searchable configuration defines how the `SearchView` behaves and is defined in a `res/xml/searchable.xml` file. At a minimum, a searchable configuration must contain an `android:label` attribute that has the same value as the `android:label` attribute of the <application> or <activity> element in your Android manifest. However, we also recommend adding an `android:hint` attribute to give the user an idea of what to enter into the search box:

```
<?xml version="1.0" encoding="utf-8"?>

<searchable
```

```
xmlns:android="http://schemas.android.com/apk/res/android"
        android:label="@string/app_name"
        android:hint="@string/search_hint" />
```

In your application's manifest file, declare a `<meta-data>` element that points to the `res/xml/searchable.xml` file, so that your application knows where to find it. Declare the element in an `<activity>` that you want to display the `SearchView` in:

```
<activity ... >
    ...
    <meta-data android:name="android.app.searchable"
            android:resource="@xml/searchable" />

</activity>
```

In the `onCreateOptionsMenu()` method that you created before, associate the searchable configuration with the `SearchView` by calling `setSearchableInfo(SearchableInfo)`:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.options_menu, menu);

    // Associate searchable configuration with the SearchView
    SearchManager searchManager =
           (SearchManager) getSystemService(Context.SEARCH_SERVICE);
    SearchView searchView =
            (SearchView) menu.findItem(R.id.search).getActionView();
    searchView.setSearchableInfo(
            searchManager.getSearchableInfo(getComponentName()));

    return true;
}
```

The call to `getSearchableInfo()` obtains a `SearchableInfo` object that is created from the searchable configuration XML file. When the searchable configuration is correctly associated with your `SearchView`, the `SearchView` starts an activity with the `ACTION_SEARCH` intent when a user submits a query. You now need an activity that can filter for this intent and handle the search query.

Create a Searchable Activity

A `SearchView` tries to start an activity with the `ACTION_SEARCH` when a user submits a search query. A searchable activity filters for the `ACTION_SEARCH`

intent and searches for the query in some sort of data set. To create a searchable activity, declare an activity of your choice to filter for the `ACTION_SEARCH` intent:

```
<activity android:name=".SearchResultsActivity" ... >
    ...
    <intent-filter>
        <action android:name="android.intent.action.SEARCH" />
    </intent-filter>
    ...
</activity>
```

In your searchable activity, handle the `ACTION_SEARCH` intent by checking for it in your `onCreate()` method.

Note: If your searchable activity launches in single top mode (`android:launchMode="singleTop"`), also handle the `ACTION_SEARCH` intent in the `onNewIntent()` method. In single top mode, only one instance of your activity is created and subsequent calls to start your activity do not create a new activity on the stack. This launch mode is useful so users can perform searches from the same activity without creating a new activity instance every time.

```
public class SearchResultsActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        ...
        handleIntent(getIntent());
    }

    @Override
    protected void onNewIntent(Intent intent) {
        ...
        handleIntent(intent);
    }

    private void handleIntent(Intent intent) {

        if (Intent.ACTION_SEARCH.equals(intent.getAction())) {
            String query =
intent.getStringExtra(SearchManager.QUERY);
            //use the query to search your data somehow
        }
    }
    ...
}
```

If you run your app now, the `SearchView` can accept the user's query and start your searchable activity with the `ACTION_SEARCH` intent. It is now up

to you to figure out how to store and search your data given a query.

Next Get started Next class

Storing and Searching for Data

Previous Next Get started

This lesson teaches you to

- Create the Virtual Table
- Populate the Virtual Table
- Search for the Query

There are many ways to store your data, such as in an online database, in a local SQLite database, or even in a text file. It is up to you to decide what is the best solution for your application. This lesson shows you how to create a SQLite virtual table that can provide robust full-text searching. The table is populated with data from a text file that contains a word and definition pair on each line in the file.

Create the Virtual Table

A virtual table behaves similarly to a SQLite table, but reads and writes to an object in memory via callbacks, instead of to a database file. To create a virtual table, create a class for the table:

```
public class DatabaseTable {
    private final DatabaseOpenHelper mDatabaseOpenHelper;

    public DatabaseTable(Context context) {
        mDatabaseOpenHelper = new DatabaseOpenHelper(context);
    }
}
```

Create an inner class in `DatabaseTable` that extends `SQLiteOpenHelper`. The `SQLiteOpenHelper` class defines abstract methods that you must override so that your database table can be created and upgraded when necessary. For example, here is some code that declares a database table that will contain words for a dictionary app:

```
public class DatabaseTable {

    private static final String TAG = "DictionaryDatabase";

    //The columns we'll include in the dictionary table
    public static final String COL_WORD = "WORD";
    public static final String COL_DEFINITION = "DEFINITION";

    private static final String DATABASE_NAME = "DICTIONARY";
    private static final String FTS_VIRTUAL_TABLE = "FTS";
    private static final int DATABASE_VERSION = 1;

    private final DatabaseOpenHelper mDatabaseOpenHelper;

    public DatabaseTable(Context context) {
        mDatabaseOpenHelper = new DatabaseOpenHelper(context);
    }

    private static class DatabaseOpenHelper extends SQLiteOpenHelper
{

        private final Context mHelperContext;
        private SQLiteDatabase mDatabase;

        private static final String FTS_TABLE_CREATE =
                    "CREATE VIRTUAL TABLE " + FTS_VIRTUAL_TABLE +
                    " USING fts3 (" +
                    COL_WORD + ", " +
                    COL_DEFINITION + ")";

        DatabaseOpenHelper(Context context) {
            super(context, DATABASE_NAME, null, DATABASE_VERSION);
            mHelperContext = context;
        }

        @Override
        public void onCreate(SQLiteDatabase db) {
            mDatabase = db;
            mDatabase.execSQL(FTS_TABLE_CREATE);
        }

        @Override
        public void onUpgrade(SQLiteDatabase db, int oldVersion, int
newVersion) {
            Log.w(TAG, "Upgrading database from version " +
oldVersion + " to "
                    + newVersion + ", which will destroy all old
data");
            db.execSQL("DROP TABLE IF EXISTS " + FTS_VIRTUAL_TABLE);
            onCreate(db);
        }
    }
}
```

Populate the Virtual Table

The table now needs data to store. The following code shows you how to

read a text file (located in `res/raw/definitions.txt`) that contains words and their definitions, how to parse that file, and how to insert each line of that file as a row in the virtual table. This is all done in another thread to prevent the UI from locking. Add the following code to your `DatabaseOpenHelper` inner class.

Tip: You also might want to set up a callback to notify your UI activity of this thread's completion.

```
private void loadDictionary() {
        new Thread(new Runnable() {
            public void run() {
                try {
                    loadWords();
                } catch (IOException e) {
                    throw new RuntimeException(e);
                }
            }
        }).start();
    }

private void loadWords() throws IOException {
    final Resources resources = mHelperContext.getResources();
    InputStream inputStream =
resources.openRawResource(R.raw.definitions);
    BufferedReader reader = new BufferedReader(new
InputStreamReader(inputStream));

    try {
        String line;
        while ((line = reader.readLine()) != null) {
            String[] strings = TextUtils.split(line, "-");
            if (strings.length < 2) continue;
            long id = addWord(strings[0].trim(), strings[1].trim());
            if (id < 0) {
                Log.e(TAG, "unable to add word: " +
strings[0].trim());
            }
        }
    } finally {
        reader.close();
    }
}

public long addWord(String word, String definition) {
    ContentValues initialValues = new ContentValues();
    initialValues.put(COL_WORD, word);
    initialValues.put(COL_DEFINITION, definition);

    return mDatabase.insert(FTS_VIRTUAL_TABLE, null, initialValues);
}
```

Call the `loadDictionary()` method wherever appropriate to populate the table. A good place would be in the `onCreate()` method of the

`DatabaseOpenHelper` class, right after you create the table:

```
@Override
public void onCreate(SQLiteDatabase db) {
    mDatabase = db;
    mDatabase.execSQL(FTS_TABLE_CREATE);
    loadDictionary();
}
```

Search for the Query

When you have the virtual table created and populated, use the query supplied by your `SearchView` to search the data. Add the following methods to the `DatabaseTable` class to build a SQL statement that searches for the query:

```
public Cursor getWordMatches(String query, String[] columns) {
    String selection = COL_WORD + " MATCH ?";
    String[] selectionArgs = new String[] {query+"*"};

    return query(selection, selectionArgs, columns);
}

private Cursor query(String selection, String[] selectionArgs,
String[] columns) {
    SQLiteQueryBuilder builder = new SQLiteQueryBuilder();
    builder.setTables(FTS_VIRTUAL_TABLE);

    Cursor cursor =
builder.query(mDatabaseOpenHelper.getReadableDatabase(),
            columns, selection, selectionArgs, null, null, null);

    if (cursor == null) {
        return null;
    } else if (!cursor.moveToFirst()) {
        cursor.close();
        return null;
    }
    return cursor;
}
```

Search for a query by calling `getWordMatches()`. Any matching results are returned in a `Cursor` that you can iterate through or use to build a `ListView`. This example calls `getWordMatches()` in the `handleIntent()` method of the searchable activity. Remember that the searchable activity receives the query inside of the `ACTION_SEARCH` intent as an extra, because of the intent filter that you previously created:

```
DatabaseTable db = new DatabaseTable(this);
```

```
...

private void handleIntent(Intent intent) {

    if (Intent.ACTION_SEARCH.equals(intent.getAction())) {
        String query = intent.getStringExtra(SearchManager.QUERY);
        Cursor c = db.getWordMatches(query, null);
        //process Cursor and display results
    }
}
```

Next Get started Next class

Remaining Backward Compatible

Previous Next Get started

This lesson teaches you to

- Set Minimum and Target API levels
- Provide the Search Dialog for Older Devices
- Check the Android Build Version at Runtime

The `SearchView` and action bar are only available on Android 3.0 and later. To support older platforms, you can fall back to the search dialog. The search dialog is a system provided UI that overlays on top of your application when invoked.

Set Minimum and Target API levels

To setup the search dialog, first declare in your manifest that you want to support older devices, but want to target Android 3.0 or later versions. When you do this, your application automatically uses the action bar on Android 3.0 or later and uses the traditional menu system on older devices:

```
<uses-sdk android:minSdkVersion="7" android:targetSdkVersion="15" />

<application>
...
```

Provide the Search Dialog for Older Devices

To invoke the search dialog on older devices, call `onSearchRequested()` whenever a user selects the search menu item from the options menu. Because Android 3.0 and higher devices show the `SearchView` in the action bar (as demonstrated in the first lesson), only versions older than

3.0 call `onOptionsItemSelected()` when the user selects the search menu item.

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.search:
            onSearchRequested();
            return true;
        default:
            return false;
    }
}
```

Check the Android Build Version at Runtime

At runtime, check the device version to make sure an unsupported use of `SearchView` does not occur on older devices. In our example code, this happens in the `onCreateOptionsMenu()` method:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {

    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.options_menu, menu);

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
        SearchManager searchManager =
                (SearchManager)
getSystemService(Context.SEARCH_SERVICE);
        SearchView searchView =
                (SearchView)
menu.findItem(R.id.search).getActionView();
        searchView.setSearchableInfo(
                searchManager.getSearchableInfo(getComponentName()));
        searchView.setIconifiedByDefault(false);
    }
    return true;
}
```

Next Get started Next class

Making Your App Content Searchable by Google

Previous Next Get started

Dependencies and prerequisites

- Android 2.3 (API level 9) and higher

You Should Also Read

- The power of Search, now across apps (blog post)
- App Indexing for Google Search
- Intents and Intent Filters

Video

DevBytes: App Indexing

As mobile apps become more pervasive, users are looking for relevant information not only from web sites but also from apps they have installed. You can enable Google to crawl through your app content and present your Android app as a destination to users through Google Search results, when that content corresponds to a web page that you own.

You can make it possible for Google Search to open specific content in your app by providing intent filters for your activities. Google Search app indexing complements this capability by presenting links to relevant app content alongside links to your web pages in users' search results. Users on mobile devices can then click on a link to open your app from their search results, allowing them to directly view your app's content instead

of a web page.

To enable Google Search app indexing, you need to provide Google with information about the relationship between your app and web site. This process involves the following steps:

1. Enable deep linking to specific content in your app by adding intent filters in your app manifest.
2. Annotate these links in the associated web pages on your web site or in a Sitemap file.
3. Opt in to allow Googlebot to crawl through your APK in the Google Play store to index your app content. You are automatically opted-in when you join as a participant in the early adopter program.

This class shows how to enable deep linking and indexing of your application content so that users can open this content directly from mobile search results.

Lessons

Enabling Deep Links for App Content

Shows how to add intent filters to enable deep linking to app content.

Specifying App Content for Indexing

Shows how to annotate web site metadata to allow Google's algorithms to index app content.

Next Get started Next class

Enabling Deep Links for App Content

Previous Next Get started

This lesson teaches you to

1. Add Intent Filters for Your Deep Links
2. Read Data from Incoming Intents
3. Test Your Deep Links

You should also read

- Intents and Intent Filters
- Allow Other Apps to Start Your Activity

To enable Google to crawl your app content and allow users to enter your app from search results, you must add intent filters for the relevant activities in your app manifest. These intent filters allow *deep linking* to the content in any of your activities. For example, the user might click on a deep link to view a page within a shopping app that describes a product offering that the user is searching for.

Add Intent Filters for Your Deep Links

To create a deep link to your app content, add an intent filter that contains these elements and attribute values in your manifest:

`<action>`
: Specify the `ACTION_VIEW` intent action so that the intent filter can be reached from Google Search.

`<data>`

Add one or more `<data>` tags, where each tag represents a URI format that resolves to the activity. At minimum, the `<data>` tag must include the `android:scheme` attribute.

You can add additional attributes to further refine the type of URI that the activity accepts. For example, you might have multiple activities that accept similar URIs, but which differ simply based on the path name. In this case, use the `android:path` attribute or its variants (`pathPattern` or `pathPrefix`) to differentiate which activity the system should open for different URI paths.

`<category>`

Include the `BROWSABLE` category. The `BROWSABLE` category is required in order for the intent filter to be accessible from a web browser. Without it, clicking a link in a browser cannot resolve to your app. The `DEFAULT` category is optional, but recommended. Without this category, the activity can be started only with an explicit intent, using your app component name.

The following XML snippet shows how you might specify an intent filter in your manifest for deep linking. The URIs `“example://gizmos”` and `“http://www.example.com/gizmos”` both resolve to this activity.

```
<activity
    android:name="com.example.android.GizmosActivity"
    android:label="@string/title_gizmos" >
    <intent-filter android:label="@string/filter_title_viewgizmos">
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />
        <!-- Accepts URIs that begin with
"http://www.example.com/gizmos” -->
        <data android:scheme="http"
              android:host="www.example.com"
              android:pathPrefix="/gizmos" />
        <!-- note that the leading "/" is required for pathPrefix--
>
        <!-- Accepts URIs that begin with "example://gizmos”
        <data android:scheme="example"
              android:host="gizmos" />
        -->
    </intent-filter>
</activity>
```

Once you've added intent filters with URIs for activity content to your app manifest, Android is able to route any `Intent` that has matching URIs to

your app at runtime.

Note: Intent filters may only contain a single `data` element for a URI pattern. Create separate intent filters to capture additional URI patterns.

To learn more about defining intent filters, see Allow Other Apps to Start Your Activity.

Read Data from Incoming Intents

Once the system starts your activity through an intent filter, you can use data provided by the `Intent` to determine what you need to render. Call the `getData()` and `getAction()` methods to retrieve the data and action associated with the incoming `Intent`. You can call these methods at any time during the lifecycle of the activity, but you should generally do so during early callbacks such as `onCreate()` or `onStart()`.

Here's a snippet that shows how to retrieve data from an `Intent`:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    Intent intent = getIntent();
    String action = intent.getAction();
    Uri data = intent.getData();
}
```

Follow these best practices to improve the user's experience:

- The deep link should take users directly to the content, without any prompts, interstitial pages, or logins. Make sure that users can see the app content even if they never previously opened the application. It is okay to prompt users on subsequent interactions or when they open the app from the Launcher. This is the same principle as the first click free experience for web sites.
- Follow the design guidance described in Navigation with Back and Up so that your app matches users' expectations for backward navigation after they enter your app through a deep link.

Test Your Deep Links

You can use the Android Debug Bridge with the activity manager (am) tool to test that the intent filter URIs you specified for deep linking resolve to the correct app activity. You can run the adb command against a device or an emulator.

The general syntax for testing an intent filter URI with adb is:

```
$ adb shell am start
        -W -a android.intent.action.VIEW
        -d <URI> <PACKAGE>
```

For example, the command below tries to view a target app activity that is associated with the specified URI.

```
$ adb shell am start
        -W -a android.intent.action.VIEW
        -d "example://gizmos" com.example.android
```

Next Get started Next class

Specifying App Content for Indexing

Previous Next Get started

This lesson teaches you to

1. Add Deep Links in Your Sitemap
2. Add Deep Links in Your Web Pages
3. Allow Google to Crawl URLs Requested By Your App

You should also read

- Webmaster Tools
- Creating Sitemaps
- Googlebot

Google's web crawling bot (Googlebot), which crawls and indexes web sites for the Google search engine, can also index content in your Android app. By opting in, you can allow Googlebot to crawl the content in the APK through the Google Play Store to index the app content. To indicate which app content you'd like Google to index, simply add link elements either to your existing Sitemap file or in the `<head>` element of each web page in your site, in the same way as you would for web pages.

The deep links that you share with Google Search must take this URI format:

```
android-app://<package_name>/<scheme>/<host_path>
```

The components that make up the URI format are:

package_name. Represents the package name for your APK as listed in the Google Play Developer Console.
- **scheme.** The URI scheme that matches your intent filter.
- **host_path.** Identifies the specific content within your application.

The following sections describe how to add a deep link URI to your Sitemap or web pages.

Add Deep Links in Your Sitemap

To annotate the deep link for Google Search app indexing in your Sitemap, use the `<xhtml:link>` tag and specify the deep link as an alternate URI.

For example, the following XML snippet shows how you might specify a link to your web page by using the `<loc>` tag, and a corresponding deep link to your Android app by using the `<xhtml:link>` tag.

```
<?xml version="1.0" encoding="UTF-8" ?>
<urlset
    xmlns="http://www.sitemaps.org/schemas/sitemap/0.9"
    xmlns:xhtml="http://www.w3.org/1999/xhtml">
    <url>
        <loc>example://gizmos</loc>
            <xhtml:link
                rel="alternate"
                href="android-
app://com.example.android/example/gizmos" />
    </url>
    ...
</urlset>
```

Add Deep Links in Your Web Pages

Instead of specifying the deep links for Google Search app indexing in your Sitemap file, you can annotate the deep links in the HTML markup of your web pages. You can do this in the `<head>` section for each web page by adding a `<link>` tag and specifying the deep link as an alternate URI.

For example, the following HTML snippet shows how you might specify the corresponding deep link in a web page that has the URL

```
example://gizmos.

<html>
<head>
    <link rel="alternate"
          href="android-app://com.example.android/example/gizmos" />
    ...
</head>
<body> ... </body>
```

Allow Google to Crawl URLs Requested By Your App

Typically, you control how Googlebot crawls publicly accessible URLs on your site by using a `robots.txt` file. When Googlebot indexes your app content, your app might make HTTP requests as part of its normal operations. However, these requests will appear to your servers as originating from Googlebot. Therefore, you must configure your server's `robots.txt` file properly to allow these requests.

For example, the following `robots.txt` directive shows how you might allow access to a specific directory in your web site (for example, `/api/`) that your app needs to access, while restricting Googlebot's access to other parts of your site.

```
User-Agent: Googlebot
Allow: /api/
Disallow: /
```

To learn more about how to modify `robots.txt` to control web crawling, see the Controlling Crawling and Indexing Getting Started guide.

Next Get started Next class

Best Practices for User Interface

These classes teach you how to build a user interface using Android layouts for all types of devices. Android provides a flexible framework for UI design that allows your app to display different layouts for different devices, create custom UI widgets, and even control aspects of the system UI outside your app's window.

Designing for Multiple Screens

Previous Next Get started

Dependencies and prerequisites

- Android 1.6 or higher (2.1+ for the sample app)
- Basic knowledge of Activities and Fragments
- Experience building an Android User Interface
- Several features require the use of the support library

You should also read

- Supporting Multiple Screens

Try it out

Download the sample app

NewsReader.zip

Android powers hundreds of device types with several different screen sizes, ranging from small phones to large TV sets. Therefore, it's important that you design your application to be compatible with all screen sizes so it's available to as many users as possible.

But being compatible with different device types is not enough. Each screen size offers different possibilities and challenges for user interaction, so in order to truly satisfy and impress your users, your application must go beyond merely *supporting* multiple screens: it must *optimize* the user experience for each screen configuration.

This class shows you how to implement a user interface that's optimized

for several screen configurations.

The code in each lesson comes from a sample application that demonstrates best practices in optimizing for multiple screens. You can download the sample (to the right) and use it as a source of reusable code for your own application.

Note: This class and the associated sample use the support library in order to use the `Fragment` APIs on versions lower than Android 3.0. You must download and add the library to your application in order to use all APIs in this class.

Lessons

Supporting Different Screen Sizes

This lesson walks you through how to design layouts that adapts several different screen sizes (using flexible dimensions for views, `RelativeLayout`, screen size and orientation qualifiers, alias filters, and nine-patch bitmaps).

Supporting Different Screen Densities

This lesson shows you how to support screens that have different pixel densities (using density-independent pixels and providing bitmaps appropriate for each density).

Implementing Adaptative UI Flows

This lesson shows you how to implement your UI flow in a way that adapts to several screen size/density combinations (run-time detection of active layout, reacting according to current layout, handling screen configuration changes).

Next Get started Next class

Supporting Different Screen Sizes

Previous Next Get started

This lesson teaches you to

1. Use "wrap_content" and "match_parent"
2. Use RelativeLayout
3. Use Size Qualifiers
4. Use the Smallest-width Qualifier
5. Use Layout Aliases
6. Use Orientation Qualifiers
7. Use Nine-patch Bitmaps

You should also read

- Supporting Multiple Screens

Try it out

Download the sample app

NewsReader.zip

This lesson shows you how to support different screen sizes by:

- Ensuring your layout can be adequately resized to fit the screen
- Providing appropriate UI layout according to screen configuration
- Ensuring the correct layout is applied to the correct screen
- Providing bitmaps that scale correctly

Use "wrap_content" and "match_parent"

To ensure that your layout is flexible and adapts to different screen sizes, you should use `"wrap_content"` and `"match_parent"` for the width and height of some view components. If you use `"wrap_content"`, the width or height of the view is

set to the minimum size necessary to fit the content within that view, while `"match_parent"` (also known as `"fill_parent"` before API level 8) makes the component expand to match the size of its parent view.

By using the `"wrap_content"` and `"match_parent"` size values instead of hard-coded sizes, your views either use only the space required for that view or expand to fill the available space, respectively. For example:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <LinearLayout android:layout_width="match_parent"
                  android:id="@+id/linearLayout1"
                  android:gravity="center"
                  android:layout_height="50dp">
        <ImageView android:id="@+id/imageView1"
                   android:layout_height="wrap_content"
                   android:layout_width="wrap_content"
                   android:src="@drawable/logo"
                   android:paddingRight="30dp"
                   android:layout_gravity="left"
                   android:layout_weight="0" />
        <View android:layout_height="wrap_content"
              android:id="@+id/view1"
              android:layout_width="wrap_content"
              android:layout_weight="1" />
        <Button android:id="@+id/categorybutton"
                android:background="@drawable/button_bg"
                android:layout_height="match_parent"
                android:layout_weight="0"
                android:layout_width="120dp"
                style="@style/CategoryButtonStyle"/>
    </LinearLayout>

    <fragment android:id="@+id/headlines"
              android:layout_height="fill_parent"

android:name="com.example.android.newsreader.HeadlinesFragment"
              android:layout_width="match_parent" />
</LinearLayout>
```

Notice how the sample uses `"wrap_content"` and `"match_parent"` for component sizes rather than specific dimensions. This allows the layout to adapt correctly to different screen sizes and orientations.

For example, this is what this layout looks like in portrait and landscape mode. Notice that the sizes of the components adapt automatically to the width and height:

Figure 1. The News Reader sample app in portrait (left) and landscape (right).

Use RelativeLayout

You can construct fairly complex layouts using nested instances of `LinearLayout` and combinations of `"wrap_content"` and `"match_parent"` sizes. However, `LinearLayout` does not allow you to precisely control the spacial relationships of child views; views in a `LinearLayout` simply line up side-by-side. If you need child views to be oriented in variations other than a straight line, a better solution is often to use a `RelativeLayout`, which allows you to specify your layout in terms of the spacial relationships between components. For instance, you can align one child view on the left side and another view on the right side of the screen.

For example:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <TextView
        android:id="@+id/label"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Type here:"/>
    <EditText
        android:id="@+id/entry"
        android:layout_width="match_parent"
```

```
        android:layout_height="wrap_content"
        android:layout_below="@id/label"/>
    <Button
        android:id="@+id/ok"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/entry"
        android:layout_alignParentRight="true"
        android:layout_marginLeft="10dp"
        android:text="OK" />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_toLeftOf="@id/ok"
        android:layout_alignTop="@id/ok"
        android:text="Cancel" />
</RelativeLayout>
```

Figure 2 shows how this layout appears on a QVGA screen.

Figure 2. Screenshot on a QVGA screen (small screen).

Figure 3 shows how it appears on a larger screen.

Figure 3. Screenshot on a WSVGA screen (large screen).

Notice that although the size of the components changed, their spatial relationships are preserved as specified by the `RelativeLayout.LayoutParams`.

Use Size Qualifiers

There's only so much mileage you can get from a flexible layout or relative layout like the one in the previous sections. While those layouts adapt to different screens by stretching the space within and around components, they may not provide the best user experience for each screen size. Therefore, your application should not only implement flexible layouts, but should also provide several alternative layouts to target different screen configurations. You do so by using configuration qualifiers, which allows the runtime to automatically select the appropriate resource based on the current device's configuration (such as a different layout design for different screen sizes).

For example, many applications implement the "two pane" pattern for large screens (the app might show a list of items on one pane and the content on another pane). Tablets and TVs are large enough for both panes to fit simultaneously on screen, but phone screens have to show them separately. So,

to implement these layouts, you could have the following files:

- `res/layout/main.xml`, single-pane (default) layout:

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <fragment android:id="@+id/headlines"
              android:layout_height="fill_parent"

android:name="com.example.android.newsreader.HeadlinesFragment"
              android:layout_width="match_parent" />
</LinearLayout>
```

- `res/layout-large/main.xml`, two-pane layout:

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal">
    <fragment android:id="@+id/headlines"
              android:layout_height="fill_parent"

android:name="com.example.android.newsreader.HeadlinesFragment"
              android:layout_width="400dp"
              android:layout_marginRight="10dp"/>
    <fragment android:id="@+id/article"
              android:layout_height="fill_parent"

android:name="com.example.android.newsreader.ArticleFragment"
              android:layout_width="fill_parent" />
</LinearLayout>
```

Notice the `large` qualifier in the directory name of the second layout. This layout will be selected on devices with screens classified as large (for example, 7" tablets and above). The other layout (without qualifiers) will be selected for smaller devices.

Use the Smallest-width Qualifier

One of the difficulties developers had in pre-3.2 Android devices was the "large" screen size bin, which encompasses the Dell Streak, the original Galaxy Tab, and 7" tablets in general. However, many applications may want to show different layouts for different devices in this category (such as for 5" and 7" devices), even though they are all considered to be "large" screens. That's why Android introduced the "Smallest-width" qualifier (amongst others) in Android 3.2.

The Smallest-width qualifier allows you to target screens that have a certain

minimum width given in dp. For example, the typical 7" tablet has a minimum width of 600 dp, so if you want your UI to have two panes on those screens (but a single list on smaller screens), you can use the same two layouts from the previous section for single and two-pane layouts, but instead of the `large` size qualifier, use `sw600dp` to indicate the two-pane layout is for screens on which the smallest-width is 600 dp:

- `res/layout/main.xml`, single-pane (default) layout:

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <fragment android:id="@+id/headlines"
              android:layout_height="fill_parent"

android:name="com.example.android.newsreader.HeadlinesFragment"
              android:layout_width="match_parent" />
</LinearLayout>
```

- `res/layout-sw600dp/main.xml`, two-pane layout:

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal">
    <fragment android:id="@+id/headlines"
              android:layout_height="fill_parent"

android:name="com.example.android.newsreader.HeadlinesFragment"
              android:layout_width="400dp"
              android:layout_marginRight="10dp"/>
    <fragment android:id="@+id/article"
              android:layout_height="fill_parent"

android:name="com.example.android.newsreader.ArticleFragment"
              android:layout_width="fill_parent" />
</LinearLayout>
```

This means that devices whose smallest width is greater than or equal to 600dp will select the `layout-sw600dp/main.xml` (two-pane) layout, while smaller screens will select the `layout/main.xml` (single-pane) layout.

However, this won't work well on pre-3.2 devices, because they don't recognize `sw600dp` as a size qualifier, so you still have to use the `large` qualifier as well. So, you should have a file named `res/layout-large/main.xml` which is identical to `res/layout-sw600dp/main.xml`. In the next section you'll see a technique that allows you to avoid duplicating the layout files this way.

Use Layout Aliases

The smallest-width qualifier is available only on Android 3.2 and above. Therefore, you should also still use the abstract size bins (small, normal, large and xlarge) to be compatible with earlier versions. For example, if you want to design your UI so that it shows a single-pane UI on phones but a multi-pane UI on 7" tablets, TVs and other large devices, you'd have to supply these files:

- `res/layout/main.xml`: single-pane layout
- `res/layout-large`: multi-pane layout
- `res/layout-sw600dp`: multi-pane layout

The last two files are identical, because one of them will be matched by Android 3.2 devices, and the other one is for the benefit of tablets and TVs with earlier versions of Android.

To avoid this duplication of the same file for tablets and TVs (and the maintenance headache resulting from it), you can use alias files. For example, you can define the following layouts:

- `res/layout/main.xml`, single-pane layout
- `res/layout/main_twopanes.xml`, two-pane layout

And add these two files:

- `res/values-large/layout.xml`:

  ```
  <resources>
      <item name="main" type="layout">@layout/main_twopanes</item>
  </resources>
  ```

- `res/values-sw600dp/layout.xml`:

  ```
  <resources>
      <item name="main" type="layout">@layout/main_twopanes</item>
  </resources>
  ```

These latter two files have identical content, but they don’t actually define the layout. They merely set up `main` to be an alias to `main_twopanes`. Since these files have `large` and `sw600dp` selectors, they are applied to tablets and TVs regardless of Android version (pre-3.2 tablets and TVs match `large`, and post-3.2 will match `sw600dp`).

Use Orientation Qualifiers

Some layouts work well in both landscape and portrait orientations, but most of

them can benefit from adjustments. In the News Reader sample app, here is how the layout behaves in each screen size and orientation:

- **small screen, portrait:** single pane, with logo
- **small screen, landscape:** single pane, with logo
- **7" tablet, portrait:** single pane, with action bar
- **7" tablet, landscape:** dual pane, wide, with action bar
- **10" tablet, portrait:** dual pane, narrow, with action bar
- **10" tablet, landscape:** dual pane, wide, with action bar
- **TV, landscape:** dual pane, wide, with action bar

So each of these layouts is defined in an XML file in the `res/layout/` directory. To then assign each layout to the various screen configurations, the app uses layout aliases to match them to each configuration:

```
res/layout/onepane.xml:

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <fragment android:id="@+id/headlines"
              android:layout_height="fill_parent"

android:name="com.example.android.newsreader.HeadlinesFragment"
              android:layout_width="match_parent" />
</LinearLayout>


res/layout/onepane_with_bar.xml:

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <LinearLayout android:layout_width="match_parent"
                  android:id="@+id/linearLayout1"
                  android:gravity="center"
                  android:layout_height="50dp">
        <ImageView android:id="@+id/imageView1"
                   android:layout_height="wrap_content"
                   android:layout_width="wrap_content"
                   android:src="@drawable/logo"
                   android:paddingRight="30dp"
                   android:layout_gravity="left"
                   android:layout_weight="0" />
        <View android:layout_height="wrap_content"
              android:id="@+id/view1"
              android:layout_width="wrap_content"
              android:layout_weight="1" />
        <Button android:id="@+id/categorybutton"
                android:background="@drawable/button_bg"
                android:layout_height="match_parent"
                android:layout_weight="0"
                android:layout_width="120dp"
                style="@style/CategoryButtonStyle"/>
```

```
    </LinearLayout>

    <fragment android:id="@+id/headlines"
              android:layout_height="fill_parent"

android:name="com.example.android.newsreader.HeadlinesFragment"
              android:layout_width="match_parent" />
</LinearLayout>
```

res/layout/twopanes.xml:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal">
    <fragment android:id="@+id/headlines"
              android:layout_height="fill_parent"

android:name="com.example.android.newsreader.HeadlinesFragment"
              android:layout_width="400dp"
              android:layout_marginRight="10dp"/>
    <fragment android:id="@+id/article"
              android:layout_height="fill_parent"
              android:name="com.example.android.newsreader.ArticleFragment"
              android:layout_width="fill_parent" />
</LinearLayout>
```

res/layout/twopanes_narrow.xml:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal">
    <fragment android:id="@+id/headlines"
              android:layout_height="fill_parent"

android:name="com.example.android.newsreader.HeadlinesFragment"
              android:layout_width="200dp"
              android:layout_marginRight="10dp"/>
    <fragment android:id="@+id/article"
              android:layout_height="fill_parent"
              android:name="com.example.android.newsreader.ArticleFragment"
              android:layout_width="fill_parent" />
</LinearLayout>
```

Now that all possible layouts are defined, it's just a matter of mapping the correct layout to each configuration using the configuration qualifiers. You can now do it using the layout alias technique:

res/values/layouts.xml:

```
<resources>
    <item name="main_layout" type="layout">@layout/onepane_with_bar</item>
    <bool name="has_two_panes">false</bool>
</resources>
```

res/values-sw600dp-land/layouts.xml:

```
<resources>
```

```
    <item name="main_layout" type="layout">@layout/twopanes</item>
    <bool name="has_two_panes">true</bool>
</resources>
```

res/values-sw600dp-port/layouts.xml:

```
<resources>
    <item name="main_layout" type="layout">@layout/onepane</item>
    <bool name="has_two_panes">false</bool>
</resources>
```

res/values-large-land/layouts.xml:

```
<resources>
    <item name="main_layout" type="layout">@layout/twopanes</item>
    <bool name="has_two_panes">true</bool>
</resources>
```

res/values-large-port/layouts.xml:

```
<resources>
    <item name="main_layout" type="layout">@layout/twopanes_narrow</item>
    <bool name="has_two_panes">true</bool>
</resources>
```

Use Nine-patch Bitmaps

Supporting different screen sizes usually means that your image resources must also be capable of adapting to different sizes. For example, a button background must fit whichever button shape it is applied to.

If you use simple images on components that can change size, you will quickly notice that the results are somewhat less than impressive, since the runtime will stretch or shrink your images uniformly. The solution is using nine-patch bitmaps, which are specially formatted PNG files that indicate which areas can and cannot be stretched.

Therefore, when designing bitmaps that will be used on components with variable size, always use nine-patches. To convert a bitmap into a nine-patch, you can start with a regular image (figure 4, shown with in 4x zoom for clarity).

Figure 4. `button.png`

And then run it through the `draw9patch` utility of the SDK (which is located in the `tools/` directory), in which you can mark the areas that should be stretched by drawing pixels along the left and top borders. You can also mark the area that should hold the content by drawing pixels along the right and bottom borders, resulting in figure 5.

Figure 5. `button.9.png`

Notice the black pixels along the borders. The ones on the top and left borders indicate the places where the image can be stretched, and the ones on the right and bottom borders indicate where the content should be placed.

Also, notice the `.9.png` extension. You must use this extension, since this is how the framework detects that this is a nine-patch image, as opposed to a regular PNG image.

When you apply this background to a component (by setting `android:background="@drawable/button"`), the framework stretches the image correctly to accommodate the size of the button, as shown in various sizes in figure 6.

Figure 6. A button using the `button.9.png` nine-patch in various sizes.

Next Get started Next class

Supporting Different Densities

Previous Next Get started

This lesson teaches you to

1. Use Density-independent Pixels
2. Provide Alternative Bitmaps

You should also read

- Supporting Multiple Screens
- Iconography
- Icon Design Guidelines

Try it out

Download the sample app

NewsReader.zip

This lesson shows you how to support different screen densities by providing different resources and using resolution-independent units of measurements.

Use Density-independent Pixels

One common pitfall you must avoid when designing your layouts is using absolute pixels to define distances or sizes. Defining layout dimensions with pixels is a problem because different screens have different pixel densities, so the same number of pixels may correspond to different physical sizes on different devices. Therefore, when specifying

dimensions, always use either `dp` or `sp` units. A `dp` is a density-independent pixel that corresponds to the physical size of a pixel at 160 dpi. An `sp` is the same base unit, but is scaled by the user's preferred text size (it's a scale-independent pixel), so you should use this measurement unit when defining text size (but never for layout sizes).

Video

DesignBytes: Density-independent Pixels

For example, when you specify spacing between two views, use `dp` rather than `px`:

```
<Button android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/clickme"
    android:layout_marginTop="20dp" />
```

When specifying text size, always use `sp`:

```
<TextView android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:textSize="20sp" />
```

Provide Alternative Bitmaps

Since Android runs in devices with a wide variety of screen densities, you should always provide your bitmap resources tailored to each of the generalized density buckets: low, medium, high and extra-high density. This will help you achieve good graphical quality and performance on all screen densities.

To generate these images, you should start with your raw resource in vector format and generate the images for each density using the following size scale:

- `xhdpi`: 2.0
- `hdpi`: 1.5
- `mdpi`: 1.0 (baseline)

- `ldpi`: 0.75

This means that if you generate a 200x200 image for `xhdpi` devices, you should generate the same resource in 150x150 for `hdpi`, 100x100 for `mdpi` and finally a 75x75 image for `ldpi` devices.

Then, place the generated image files in the appropriate subdirectory under `res/` and the system will pick the correct one automatically based on the screen density of the device your application is running on:

```
MyProject/
  res/
    drawable-xhdpi/
        awesomeimage.png
    drawable-hdpi/
        awesomeimage.png
    drawable-mdpi/
        awesomeimage.png
    drawable-ldpi/
        awesomeimage.png
```

Then, any time you reference `@drawable/awesomeimage`, the system selects the appropriate bitmap based on the screen's dpi.

Place your launcher icons in the `mipmap/` folders.

```
res/...
    mipmap-ldpi/...
        finished_launcher_asset.png
    mipmap-mdpi/...
        finished_launcher_asset.png
    mipmap-hdpi/...
        finished_launcher_asset.png
    mipmap-xhdpi/...
        finished_launcher_asset.png
    mipmap-xxhdpi/...
        finished_launcher_asset.png
    mipmap-xxxhdpi/...
        finished_launcher_asset.png
```

Note: You should place all launcher icons in the `res/mipmap-[density]/` folders, rather than `drawable/` folders to ensure launcher apps use the best resolution icon. For more information about using the mipmap folders, see Managing Projects Overview.

For more tips and guidelines for creating icon assets for your application, see the Icon Design Guidelines.

Next Get started Next class

Implementing Adaptative UI Flows

Previous Next Get started

This lesson teaches you to

1. Determine the Current Layout
2. React According to Current Layout
3. Reuse Fragments in Other Activities
4. Handle Screen Configuration Changes

You should also read

- Supporting Tablets and Handsets

Try it out

Download the sample app

NewsReader.zip

Depending on the layout that your application is currently showing, the UI flow may be different. For example, if your application is in the dual-pane mode, clicking on an item on the left pane will simply display the content on the right pane; if it is in single-pane mode, the content should be displayed on its own (in a different activity).

Determine the Current Layout

Since your implementation of each layout will be a little different, one of the first things you will probably have to do is determine what layout the

user is currently viewing. For example, you might want to know whether the user is in "single pane" mode or "dual pane" mode. You can do that by querying if a given view exists and is visible:

```
public class NewsReaderActivity extends FragmentActivity {
    boolean mIsDualPane;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main_layout);

        View articleView = findViewById(R.id.article);
        mIsDualPane = articleView != null &&
                        articleView.getVisibility() == View.VISIBLE;
    }
}
```

Notice that this code queries whether the "article" pane is available or not, which is much more flexible than hard-coding a query for a specific layout.

Another example of how you can adapt to the existence of different components is to check whether they are available before performing an operation on them. For example, in the News Reader sample app, there is a button that opens a menu, but that button only exists when running on versions older than Android 3.0 (because it's function is taken over by the `ActionBar` on API level 11+). So, to add the event listener for this button, you can do:

```
Button catButton = (Button) findViewById(R.id.categorybutton);
OnClickListener listener = /* create your listener here */;
if (catButton != null) {
    catButton.setOnClickListener(listener);
}
```

React According to Current Layout

Some actions may have a different result depending on the current layout. For example, in the News Reader sample, clicking on a headline from the headlines list opens the article in the right hand-side pane if the UI is in dual pane mode, but will launch a separate activity if the UI is in single-pane mode:

```
@Override
```

```
public void onHeadlineSelected(int index) {
    mArtIndex = index;
    if (mIsDualPane) {
        /* display article on the right pane */

mArticleFragment.displayArticle(mCurrentCat.getArticle(index));
    } else {
        /* start a separate activity */
        Intent intent = new Intent(this, ArticleActivity.class);
        intent.putExtra("catIndex", mCatIndex);
        intent.putExtra("artIndex", index);
        startActivity(intent);
    }
}
```

Likewise, if the app is in dual-pane mode, it should set up the action bar with tabs for navigation, whereas if the app is in single-pane mode, it should set up navigation with a spinner widget. So your code should also check which case is appropriate:

```
final String CATEGORIES[] = { "Top Stories", "Politics", "Economy",
"Technology" };

public void onCreate(Bundle savedInstanceState) {
    ....
    if (mIsDualPane) {
        /* use tabs for navigation */

actionBar.setNavigationMode(android.app.ActionBar.NAVIGATION_MODE_TABS)

        int i;
        for (i = 0; i < CATEGORIES.length; i++) {
            actionBar.addTab(actionBar.newTab().setText(
                CATEGORIES[i]).setTabListener(handler));
        }
        actionBar.setSelectedNavigationItem(selTab);
    }
    else {
        /* use list navigation (spinner) */

actionBar.setNavigationMode(android.app.ActionBar.NAVIGATION_MODE_LIST)

        SpinnerAdapter adap = new ArrayAdapter(this,
                R.layout.headline_item, CATEGORIES);
        actionBar.setListNavigationCallbacks(adap, handler);
    }
}
```

Reuse Fragments in Other Activities

A recurring pattern in designing for multiple screens is having a portion of your interface that's implemented as a pane on some screen configurations and as a separate activity on other configurations. For

example, in the News Reader sample, the news article text is presented in the right side pane on large screens, but is a separate activity on smaller screens.

In cases like this, you can usually avoid code duplication by reusing the same `Fragment` subclass in several activities. For example, `ArticleFragment` is used in the dual-pane layout:

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal">
    <fragment android:id="@+id/headlines"
              android:layout_height="fill_parent"

android:name="com.example.android.newsreader.HeadlinesFragment"
              android:layout_width="400dp"
              android:layout_marginRight="10dp"/>
    <fragment android:id="@+id/article"
              android:layout_height="fill_parent"

android:name="com.example.android.newsreader.ArticleFragment"
              android:layout_width="fill_parent" />
</LinearLayout>
```

And reused (without a layout) in the activity layout for smaller screens (`ArticleActivity`):

```
ArticleFragment frag = new ArticleFragment();
getSupportFragmentManager().beginTransaction().add(android.R.id.content
frag).commit();
```

Naturally, this has the same effect as declaring the fragment in an XML layout, but in this case an XML layout is unnecessary work because the article fragment is the only component of this activity.

One very important point to keep in mind when designing your fragments is to not create a strong coupling to a specific activity. You can usually do that by defining an interface that abstracts all the ways in which the fragment needs to interact with its host activity, and then the host activity implements that interface:

For example, the News Reader app's `HeadlinesFragment` does precisely that:

```
public class HeadlinesFragment extends ListFragment {
```

```
    ...
    OnHeadlineSelectedListener mHeadlineSelectedListener = null;

    /* Must be implemented by host activity */
    public interface OnHeadlineSelectedListener {
        public void onHeadlineSelected(int index);
    }
    ...

    public void
setOnHeadlineSelectedListener(OnHeadlineSelectedListener listener) {
        mHeadlineSelectedListener = listener;
    }
}
```

Then, when the user selects a headline, the fragment notifies the listener specified by the host activity (as opposed to notifying a specific hard-coded activity):

```
public class HeadlinesFragment extends ListFragment {
    ...
    @Override
    public void onItemClick(AdapterView<?> parent,
                            View view, int position, long id) {
        if (null != mHeadlineSelectedListener) {
            mHeadlineSelectedListener.onHeadlineSelected(position);
        }
    }
    ...
}
```

This technique is discussed further in the guide to Supporting Tablets and Handsets.

Handle Screen Configuration Changes

If you are using separate activities to implement separate parts of your interface, you have to keep in mind that it may be necessary to react to certain configuration changes (such as a rotation change) in order to keep your interface consistent.

For example, on a typical 7" tablet running Android 3.0 or higher, the News Reader sample uses a separate activity to display the news article when running in portrait mode, but uses a two-pane layout when in landscape mode.

This means that when the user is in portrait mode and the activity for

viewing an article is onscreen, you need to detect that the orientation changed to landscape and react appropriately by ending the activity and return to the main activity so the content can display in the two-pane layout:

```
public class ArticleActivity extends FragmentActivity {
    int mCatIndex, mArtIndex;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mCatIndex = getIntent().getExtras().getInt("catIndex", 0);
        mArtIndex = getIntent().getExtras().getInt("artIndex", 0);

        // If should be in two-pane mode, finish to return to main
activity
        if (getResources().getBoolean(R.bool.has_two_panes)) {
            finish();
            return;
        }
        ...
}
```

Next Get started Next class

Creating Custom Views

Previous Next Get started

Dependencies and prerequisites

- Android 2.1 (API level 7) or higher

You should also read

- Custom Components
- Input Events
- Property Animation
- Hardware Acceleration
- Accessibility developer guide

Try it out

Download the sample

CustomView.zip

The Android framework has a large set of `View` classes for interacting with the user and displaying various types of data. But sometimes your app has unique needs that aren't covered by the built-in views. This class shows you how to create your own views that are robust and reusable.

Lessons

Creating a View Class

Create a class that acts like a built-in view, with custom attributes and support from the ADT layout editor.

Custom Drawing

Make your view visually distinctive using the Android graphics system.

Making the View Interactive

Users expect a view to react smoothly and naturally to input gestures. This lesson discusses how to use gesture detection, physics, and animation to give your user interface a professional feel.

Optimizing the View

No matter how beautiful your UI is, users won't love it if it doesn't run at a consistently high frame rate. Learn how to avoid common performance problems, and how to use hardware acceleration to make your custom drawings run faster.

Next Get started Next class

Creating a View Class

Previous Next Get started

This lesson teaches you to

1. Subclass a View
2. Define Custom Attributes
3. Apply Custom Attributes to a View
4. Add Properties and Events
5. Design For Accessibility

You should also read

- Custom Components

Try it out

Download the sample

CustomView.zip

A well-designed custom view is much like any other well-designed class. It encapsulates a specific set of functionality with an easy to use interface, it uses CPU and memory efficiently, and so forth. In addition to being a well-designed class, though, a custom view should:

- Conform to Android standards
- Provide custom styleable attributes that work with Android XML layouts
- Send accessibility events
- Be compatible with multiple Android platforms.

The Android framework provides a set of base classes and XML tags to help you create a view that meets all of these requirements. This lesson discusses how to use the Android framework to create the core functionality of a view class.

Subclass a View

All of the view classes defined in the Android framework extend `View`. Your custom view can also extend `View` directly, or you can save time by extending one of the existing view subclasses, such as `Button`.

To allow the Android Developer Tools to interact with your view, at a minimum you must provide a constructor that takes a `Context` and an `AttributeSet` object as parameters. This constructor allows the layout editor to create and edit an instance of your view.

```
class PieChart extends View {
    public PieChart(Context context, AttributeSet attrs) {
        super(context, attrs);
    }
}
```

Define Custom Attributes

To add a built-in `View` to your user interface, you specify it in an XML element and control its appearance and behavior with element attributes. Well-written custom views can also be added and styled via XML. To enable this behavior in your custom view, you must:

- Define custom attributes for your view in a `<declare-styleable>` resource element
- Specify values for the attributes in your XML layout
- Retrieve attribute values at runtime
- Apply the retrieved attribute values to your view

This section discusses how to define custom attributes and specify their values. The next section deals with retrieving and applying the values at runtime.

To define custom attributes, add `<declare-styleable>` resources to your project. It's customary to put these resources into a `res/values/attrs.xml` file. Here's an example of an `attrs.xml` file:

```
<resources>
   <declare-styleable name="PieChart">
       <attr name="showText" format="boolean" />
       <attr name="labelPosition" format="enum">
           <enum name="left" value="0"/>
           <enum name="right" value="1"/>
       </attr>
   </declare-styleable>
</resources>
```

This code declares two custom attributes, `showText` and `labelPosition`, that belong to a styleable entity named `PieChart`. The name of the styleable entity is, by convention, the same name as the name of the class that defines the custom view. Although it's not strictly necessary to follow this convention, many popular code editors depend on this naming convention to provide statement completion.

Once you define the custom attributes, you can use them in layout XML files just like built-in attributes. The only difference is that your custom attributes belong to a different namespace. Instead of belonging to the `http://schemas.android.com/apk/res/android` namespace, they belong to `http://schemas.android.com/apk/res/[your package name]`. For example, here's how to use the attributes defined for `PieChart`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:custom="http://schemas.android.com/apk/res/com.example.customview

 <com.example.customviews.charting.PieChart
     custom:showText="true"
     custom:labelPosition="left" />
</LinearLayout>
```

In order to avoid having to repeat the long namespace URI, the sample uses an `xmlns` directive. This directive assigns the alias `custom` to the namespace `http://schemas.android.com/apk/res/com.example.customviews`. You can choose any alias you want for your namespace.

Notice the name of the XML tag that adds the custom view to the layout.

It is the fully qualified name of the custom view class. If your view class is an inner class, you must further qualify it with the name of the view's outer class. further. For instance, the `PieChart` class has an inner class called `PieView`. To use the custom attributes from this class, you would use the tag `com.example.customviews.charting.PieChart$PieView`.

Apply Custom Attributes

When a view is created from an XML layout, all of the attributes in the XML tag are read from the resource bundle and passed into the view's constructor as an `AttributeSet`. Although it's possible to read values from the `AttributeSet` directly, doing so has some disadvantages:

- Resource references within attribute values are not resolved
- Styles are not applied

Instead, pass the `AttributeSet` to `obtainStyledAttributes()`. This method passes back a `TypedArray` array of values that have already been dereferenced and styled.

The Android resource compiler does a lot of work for you to make calling `obtainStyledAttributes()` easier. For each `<declare-styleable>` resource in the res directory, the generated R.java defines both an array of attribute ids and a set of constants that define the index for each attribute in the array. You use the predefined constants to read the attributes from the `TypedArray`. Here's how the `PieChart` class reads its attributes:

```
public PieChart(Context context, AttributeSet attrs) {
   super(context, attrs);
   TypedArray a = context.getTheme().obtainStyledAttributes(
        attrs,
        R.styleable.PieChart,
        0, 0);

   try {
       mShowText = a.getBoolean(R.styleable.PieChart_showText,
false);
       mTextPos = a.getInteger(R.styleable.PieChart_labelPosition,
0);
   } finally {
       a.recycle();
   }
}
```

Note that `TypedArray` objects are a shared resource and must be recycled after use.

Add Properties and Events

Attributes are a powerful way of controlling the behavior and appearance of views, but they can only be read when the view is initialized. To provide dynamic behavior, expose a property getter and setter pair for each custom attribute. The following snippet shows how `PieChart` exposes a property called `showText`:

```
public boolean isShowText() {
   return mShowText;
}

public void setShowText(boolean showText) {
   mShowText = showText;
   invalidate();
   requestLayout();
}
```

Notice that `setShowText` calls `invalidate()` and `requestLayout()`. These calls are crucial to ensure that the view behaves reliably. You have to invalidate the view after any change to its properties that might change its appearance, so that the system knows that it needs to be redrawn. Likewise, you need to request a new layout if a property changes that might affect the size or shape of the view. Forgetting these method calls can cause hard-to-find bugs.

Custom views should also support event listeners to communicate important events. For instance, `PieChart` exposes a custom event called `OnCurrentItemChanged` to notify listeners that the user has rotated the pie chart to focus on a new pie slice.

It's easy to forget to expose properties and events, especially when you're the only user of the custom view. Taking some time to carefully define your view's interface reduces future maintenance costs. A good rule to follow is to always expose any property that affects the visible appearance or behavior of your custom view.

Design For Accessibility

Your custom view should support the widest range of users. This includes users with disabilities that prevent them from seeing or using a touchscreen. To support users with disabilities, you should:

- Label your input fields using the `android:contentDescription` attribute
- Send accessibility events by calling `sendAccessibilityEvent()` when appropriate.
- Support alternate controllers, such as D-pad and trackball

For more information on creating accessible views, see Making Applications Accessible in the Android Developers Guide.

Next Get started Next class

Custom Drawing

Previous Next Get started

This lesson teaches you to

1. Override onDraw()
2. Create Drawing Objects
3. Handle Layout Events
4. Draw!

You should also read

- Canvas and Drawables

Try it out

Download the sample

CustomView.zip

The most important part of a custom view is its appearance. Custom drawing can be easy or complex according to your application's needs. This lesson covers some of the most common operations.

Override onDraw()

The most important step in drawing a custom view is to override the `onDraw()` method. The parameter to `onDraw()` is a `Canvas` object that the view can use to draw itself. The `Canvas` class defines methods for drawing text, lines, bitmaps, and many other graphics primitives. You can use these methods in `onDraw()` to create your custom user interface (UI).

Before you can call any drawing methods, though, it's necessary to create a `Paint` object. The next section discusses `Paint` in more detail.

Create Drawing Objects

The `android.graphics` framework divides drawing into two areas:

- *What* to draw, handled by `Canvas`
- *How* to draw, handled by `Paint`.

For instance, `Canvas` provides a method to draw a line, while `Paint` provides methods to define that line's color. `Canvas` has a method to draw a rectangle, while `Paint` defines whether to fill that rectangle with a color or leave it empty. Simply put, `Canvas` defines shapes that you can draw on the screen, while `Paint` defines the color, style, font, and so forth of each shape you draw.

So, before you draw anything, you need to create one or more `Paint` objects. The `PieChart` example does this in a method called `init`, which is called from the constructor:

```
private void init() {
   mTextPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
   mTextPaint.setColor(mTextColor);
   if (mTextHeight == 0) {
       mTextHeight = mTextPaint.getTextSize();
   } else {
       mTextPaint.setTextSize(mTextHeight);
   }

   mPiePaint = new Paint(Paint.ANTI_ALIAS_FLAG);
   mPiePaint.setStyle(Paint.Style.FILL);
   mPiePaint.setTextSize(mTextHeight);

   mShadowPaint = new Paint(0);
   mShadowPaint.setColor(0xff101010);
   mShadowPaint.setMaskFilter(new BlurMaskFilter(8,
BlurMaskFilter.Blur.NORMAL));

   ...
```

Creating objects ahead of time is an important optimization. Views are redrawn very frequently, and many drawing objects require expensive initialization. Creating drawing objects within your `onDraw()` method significantly reduces performance and can make your UI appear sluggish.

Handle Layout Events

In order to properly draw your custom view, you need to know what size it is. Complex custom views often need to perform multiple layout calculations depending on the size and shape of their area on screen. You should never make assumptions about the size of your view on the screen. Even if only one app uses your view, that app needs to handle different screen sizes, multiple screen densities, and various aspect ratios in both portrait and landscape mode.

Although `View` has many methods for handling measurement, most of them do not need to be overridden. If your view doesn't need special control over its size, you only need to override one method: `onSizeChanged()`.

`onSizeChanged()` is called when your view is first assigned a size, and again if the size of your view changes for any reason. Calculate positions, dimensions, and any other values related to your view's size in `onSizeChanged()`, instead of recalculating them every time you draw. In the `PieChart` example, `onSizeChanged()` is where the `PieChart` view calculates the bounding rectangle of the pie chart and the relative position of the text label and other visual elements.

When your view is assigned a size, the layout manager assumes that the size includes all of the view's padding. You must handle the padding values when you calculate your view's size. Here's a snippet from `PieChart.onSizeChanged()` that shows how to do this:

```
// Account for padding
float xpad = (float)(getPaddingLeft() + getPaddingRight());
float ypad = (float)(getPaddingTop() + getPaddingBottom());

// Account for the label
if (mShowText) xpad += mTextWidth;

float ww = (float)w - xpad;
float hh = (float)h - ypad;

// Figure out how big we can make the pie.
float diameter = Math.min(ww, hh);
```

If you need finer control over your view's layout parameters, implement

onMeasure(). This method's parameters are View.MeasureSpec values that tell you how big your view's parent wants your view to be, and whether that size is a hard maximum or just a suggestion. As an optimization, these values are stored as packed integers, and you use the static methods of View.MeasureSpec to unpack the information stored in each integer.

Here's an example implementation of onMeasure(). In this implementation, PieChart attempts to make its area big enough to make the pie as big as its label:

```
@Override
protected void onMeasure(int widthMeasureSpec, int
heightMeasureSpec) {
   // Try for a width based on our minimum
   int minw = getPaddingLeft() + getPaddingRight() +
getSuggestedMinimumWidth();
   int w = resolveSizeAndState(minw, widthMeasureSpec, 1);

   // Whatever the width ends up being, ask for a height that would
let the pie
   // get as big as it can
   int minh = MeasureSpec.getSize(w) - (int)mTextWidth +
getPaddingBottom() + getPaddingTop();
   int h = resolveSizeAndState(MeasureSpec.getSize(w) -
(int)mTextWidth, heightMeasureSpec, 0);

   setMeasuredDimension(w, h);
}
```

There are three important things to note in this code:

- The calculations take into account the view's padding. As mentioned earlier, this is the view's responsibility.
- The helper method resolveSizeAndState() is used to create the final width and height values. This helper returns an appropriate View.MeasureSpec value by comparing the view's desired size to the spec passed into onMeasure().
- onMeasure() has no return value. Instead, the method communicates its results by calling setMeasuredDimension(). Calling this method is mandatory. If you omit this call, the View class throws a runtime exception.

Draw!

Once you have your object creation and measuring code defined, you can implement `onDraw()`. Every view implements `onDraw()` differently, but there are some common operations that most views share:

- Draw text using `drawText()`. Specify the typeface by calling `setTypeface()`, and the text color by calling `setColor()`.
- Draw primitive shapes using `drawRect()`, `drawOval()`, and `drawArc()`. Change whether the shapes are filled, outlined, or both by calling `setStyle()`.
- Draw more complex shapes using the `Path` class. Define a shape by adding lines and curves to a `Path` object, then draw the shape using `drawPath()`. Just as with primitive shapes, paths can be outlined, filled, or both, depending on the `setStyle()`.
- Define gradient fills by creating `LinearGradient` objects. Call `setShader()` to use your `LinearGradient` on filled shapes.
- Draw bitmaps using `drawBitmap()`.

For example, here's the code that draws `PieChart`. It uses a mix of text, lines, and shapes.

```
protected void onDraw(Canvas canvas) {
   super.onDraw(canvas);

   // Draw the shadow
   canvas.drawOval(
           mShadowBounds,
           mShadowPaint
   );

   // Draw the label text
   canvas.drawText(mData.get(mCurrentItem).mLabel, mTextX, mTextY,
mTextPaint);

   // Draw the pie slices
   for (int i = 0; i < mData.size(); ++i) {
       Item it = mData.get(i);
       mPiePaint.setShader(it.mShader);
       canvas.drawArc(mBounds,
               360 - it.mEndAngle,
               it.mEndAngle - it.mStartAngle,
               true, mPiePaint);
   }

   // Draw the pointer
   canvas.drawLine(mTextX, mPointerY, mPointerX, mPointerY,
mTextPaint);
   canvas.drawCircle(mPointerX, mPointerY, mPointerSize, mTextPaint);
}
```

Next Get started Next class

Making the View Interactive

Previous Next Get started

This lesson teaches you to

1. Handle Input Gestures
2. Create Physically Plausible Motion
3. Make Your Transitions Smooth

You should also read

- Input Events
- Property Animation

Try it out

Download the sample

CustomView.zip

Drawing a UI is only one part of creating a custom view. You also need to make your view respond to user input in a way that closely resembles the real-world action you're mimicking. Objects should always act in the same way that real objects do. For example, images should not immediately pop out of existence and reappear somewhere else, because objects in the real world don't do that. Instead, images should move from one place to another.

Users also sense subtle behavior or feel in an interface, and react best to subtleties that mimic the real world. For example, when users fling a UI object, they should sense friction at the beginning that delays the motion, and then at the end sense momentum that carries the motion beyond the

fling.

This lesson demonstrates how to use features of the Android framework to add these real-world behaviors to your custom view.

Handle Input Gestures

Like many other UI frameworks, Android supports an input event model. User actions are turned into events that trigger callbacks, and you can override the callbacks to customize how your application responds to the user. The most common input event in the Android system is *touch*, which triggers `onTouchEvent(android.view.MotionEvent)`. Override this method to handle the event:

```
@Override
public boolean onTouchEvent(MotionEvent event) {
 return super.onTouchEvent(event);
}
```

Touch events by themselves are not particularly useful. Modern touch UIs define interactions in terms of gestures such as tapping, pulling, pushing, flinging, and zooming. To convert raw touch events into gestures, Android provides `GestureDetector`.

Construct a `GestureDetector` by passing in an instance of a class that implements `GestureDetector.OnGestureListener`. If you only want to process a few gestures, you can extend `GestureDetector.SimpleOnGestureListener` instead of implementing the `GestureDetector.OnGestureListener` interface. For instance, this code creates a class that extends `GestureDetector.SimpleOnGestureListener` and overrides `onDown(MotionEvent)`.

```
class mListener extends GestureDetector.SimpleOnGestureListener {
   @Override
   public boolean onDown(MotionEvent e) {
       return true;
   }
}
mDetector = new GestureDetector(PieChart.this.getContext(), new
mListener());
```

Whether or not you use `GestureDetector.SimpleOnGestureListener`, you

must always implement an `onDown()` method that returns `true`. This step is necessary because all gestures begin with an `onDown()` message. If you return `false` from `onDown()`, as `GestureDetector.SimpleOnGestureListener` does, the system assumes that you want to ignore the rest of the gesture, and the other methods of `GestureDetector.OnGestureListener` never get called. The only time you should return `false` from `onDown()` is if you truly want to ignore an entire gesture. Once you've implemented `GestureDetector.OnGestureListener` and created an instance of `GestureDetector`, you can use your `GestureDetector` to interpret the touch events you receive in `onTouchEvent()`.

```
@Override
public boolean onTouchEvent(MotionEvent event) {
   boolean result = mDetector.onTouchEvent(event);
   if (!result) {
       if (event.getAction() == MotionEvent.ACTION_UP) {
           stopScrolling();
           result = true;
       }
   }
   return result;
}
```

When you pass `onTouchEvent()` a touch event that it doesn't recognize as part of a gesture, it returns `false`. You can then run your own custom gesture-detection code.

Create Physically Plausible Motion

Gestures are a powerful way to control touchscreen devices, but they can be counterintuitive and difficult to remember unless they produce physically plausible results. A good example of this is the *fling* gesture, where the user quickly moves a finger across the screen and then lifts it. This gesture makes sense if the UI responds by moving quickly in the direction of the fling, then slowing down, as if the user had pushed on a flywheel and set it spinning.

However, simulating the feel of a flywheel isn't trivial. A lot of physics and math are required to get a flywheel model working correctly. Fortunately, Android provides helper classes to simulate this and other behaviors. The `Scroller` class is the basis for handling flywheel-style *fling*

gestures.

To start a fling, call `fling()` with the starting velocity and the minimum and maximum x and y values of the fling. For the velocity value, you can use the value computed for you by `GestureDetector`.

```
@Override
public boolean onFling(MotionEvent e1, MotionEvent e2, float
velocityX, float velocityY) {
   mScroller.fling(currentX, currentY, velocityX / SCALE, velocityY
/ SCALE, minX, minY, maxX, maxY);
   postInvalidate();
}
```

Note: Although the velocity calculated by `GestureDetector` is physically accurate, many developers feel that using this value makes the fling animation too fast. It's common to divide the x and y velocity by a factor of 4 to 8.

The call to `fling()` sets up the physics model for the fling gesture. Afterwards, you need to update the `Scroller` by calling `Scroller.computeScrollOffset()` at regular intervals. `computeScrollOffset()` updates the `Scroller` object's internal state by reading the current time and using the physics model to calculate the x and y position at that time. Call `getCurrX()` and `getCurrY()` to retrieve these values.

Most views pass the `Scroller` object's x and y position directly to `scrollTo()`. The PieChart example is a little different: it uses the current scroll y position to set the rotational angle of the chart.

```
if (!mScroller.isFinished()) {
    mScroller.computeScrollOffset();
    setPieRotation(mScroller.getCurrY());
}
```

The `Scroller` class computes scroll positions for you, but it does not automatically apply those positions to your view. It's your responsibility to make sure you get and apply new coordinates often enough to make the scrolling animation look smooth. There are two ways to do this:

- Call `postInvalidate()` after calling `fling()`, in order to force a redraw. This technique requires that you compute scroll offsets in

onDraw() and call postInvalidate() every time the scroll offset changes.
- Set up a ValueAnimator to animate for the duration of the fling, and add a listener to process animation updates by calling addUpdateListener().

The PieChart example uses the second approach. This technique is slightly more complex to set up, but it works more closely with the animation system and doesn't require potentially unnecessary view invalidation. The drawback is that ValueAnimator is not available prior to API level 11, so this technique cannot be used on devices running Android versions lower than 3.0.

Note: ValueAnimator isn't available prior to API level 11, but you can still use it in applications that target lower API levels. You just need to make sure to check the current API level at runtime, and omit the calls to the view animation system if the current level is less than 11.

```
        mScroller = new Scroller(getContext(), null, true);
        mScrollAnimator = ValueAnimator.ofFloat(0,1);
        mScrollAnimator.addUpdateListener(new
ValueAnimator.AnimatorUpdateListener() {
            @Override
            public void onAnimationUpdate(ValueAnimator valueAnimator)
{
                if (!mScroller.isFinished()) {
                    mScroller.computeScrollOffset();
                    setPieRotation(mScroller.getCurrY());
                } else {
                    mScrollAnimator.cancel();
                    onScrollFinished();
                }
            }
        });
```

Make Your Transitions Smooth

Users expect a modern UI to transition smoothly between states. UI elements fade in and out instead of appearing and disappearing. Motions begin and end smoothly instead of starting and stopping abruptly. The Android property animation framework, introduced in Android 3.0, makes smooth transitions easy.

To use the animation system, whenever a property changes that will

affect your view's appearance, do not change the property directly. Instead, use `ValueAnimator` to make the change. In the following example, modifying the currently selected pie slice in PieChart causes the entire chart to rotate so that the selection pointer is centered in the selected slice. `ValueAnimator` changes the rotation over a period of several hundred milliseconds, rather than immediately setting the new rotation value.

```
mAutoCenterAnimator = ObjectAnimator.ofInt(PieChart.this,
"PieRotation", 0);
mAutoCenterAnimator.setIntValues(targetAngle);
mAutoCenterAnimator.setDuration(AUTOCENTER_ANIM_DURATION);
mAutoCenterAnimator.start();
```

If the value you want to change is one of the base `View` properties, doing the animation is even easier, because Views have a built-in `ViewPropertyAnimator` that is optimized for simultaneous animation of multiple properties. For example:

```
animate().rotation(targetAngle).setDuration(ANIM_DURATION).start();
```

Next Get started Next class

Optimizing the View

Previous Next Get started

This lesson teaches you to

- Do Less, Less Frequently

Try it out

Download the sample

CustomView.zip

Now that you have a well-designed view that responds to gestures and transitions between states, ensure that the view runs fast. To avoid a UI that feels sluggish or stutters during playback, ensure that animations consistently run at 60 frames per second.

Do Less, Less Frequently

To speed up your view, eliminate unnecessary code from routines that are called frequently. Start by working on `onDraw()`, which will give you the biggest payback. In particular you should eliminate allocations in `onDraw()`, because allocations may lead to a garbage collection that would cause a stutter. Allocate objects during initialization, or between animations. Never make an allocation while an animation is running.

In addition to making `onDraw()` leaner, also make sure it's called as infrequently as possible. Most calls to `onDraw()` are the result of a call to `invalidate()`, so eliminate unnecessary calls to `invalidate()`.

Another very expensive operation is traversing layouts. Any time a view

calls `requestLayout()`, the Android UI system needs to traverse the entire view hierarchy to find out how big each view needs to be. If it finds conflicting measurements, it may need to traverse the hierarchy multiple times. UI designers sometimes create deep hierarchies of nested `ViewGroup` objects in order to get the UI to behave properly. These deep view hierarchies cause performance problems. Make your view hierarchies as shallow as possible.

If you have a complex UI, consider writing a custom `ViewGroup` to perform its layout. Unlike the built-in views, your custom view can make application-specific assumptions about the size and shape of its children, and thus avoid traversing its children to calculate measurements. The PieChart example shows how to extend `ViewGroup` as part of a custom view. PieChart has child views, but it never measures them. Instead, it sets their sizes directly according to its own custom layout algorithm.

Next Get started Next class

Creating Backward-Compatible UIs

Previous Next Get started

Dependencies and prerequisites

- API level 5
- The Android Support Package

You should also read

- ActionBarCompat
- How to have your (Cup)cake and eat it too

Try it out

Download the sample app

TabCompat.zip

This class demonstrates how to use UI components and APIs available in newer versions of Android in a backward-compatible way, ensuring that your application still runs on previous versions of the platform.

Throughout this class, the new Action Bar Tabs feature introduced in Android 3.0 (API level 11) serves as the guiding example, but you can apply these techniques to other UI components and API features.

Lessons

Abstracting the New APIs

Determine which features and APIs your application needs. Learn how to define application-specific, intermediary Java interfaces that abstract the implementation of the UI component to your application.

Proxying to the New APIs

Learn how to create an implementation of your interface that uses newer APIs.

Creating an Implementation with Older APIs

Learn how to create a custom implementation of your interface that uses older APIs.

Using the Version-Aware Component

Learn how to choose an implementation to use at runtime, and begin using the interface in your application.

Next Get started Next class

Abstracting the New APIs

Previous Next Get started

This lesson teaches you to:

- Prepare for Abstraction
- Create an Abstract Tab Interface
- Abstract ActionBar.Tab
- Abstract ActionBar Tab Methods

You should also read

- Action Bar
- Action Bar Tabs

Try it out

Download the sample app

TabCompat.zip

Suppose you want to use action bar tabs as the primary form of top-level navigation in your application. Unfortunately, the `ActionBar` APIs are only available in Android 3.0 or later (API level 11+). Thus, if you want to distribute your application to devices running earlier versions of the platform, you need to provide an implementation that supports the newer API while providing a fallback mechanism that uses older APIs.

In this class, you build a tabbed user interface (UI) component that uses abstract classes with version-specific implementations to provide backward-compatibility. This lesson describes how to create an abstraction layer for the new tab APIs as the first step toward building

the tab component.

Prepare for Abstraction

Abstraction in the Java programming language involves the creation of one or more interfaces or abstract classes to hide implementation details. In the case of newer Android APIs, you can use abstraction to build version-aware components that use the current APIs on newer devices, and fallback to older, more compatible APIs on older devices.

When using this approach, you first determine what newer classes you want to be able to use in a backward compatible way, then create abstract classes, based on the public interfaces of the newer classes. In defining the abstraction interfaces, you should mirror the newer API as much as possible. This maximizes forward-compatibility and makes it easier to drop the abstraction layer in the future when it is no longer necessary.

After creating abstract classes for these new APIs, any number of implementations can be created and chosen at runtime. For the purposes of backward-compatibility, these implementations can vary by required API level. Thus, one implementation may use recently released APIs, while others can use older APIs.

Create an Abstract Tab Interface

In order to create a backward-compatible version of tabs, you should first determine which features and specific APIs your application requires. In the case of top-level section tabs, suppose you have the following functional requirements:

1. Tab indicators should show text and an icon.
2. Tabs can be associated with a fragment instance.
3. The activity should be able to listen for tab changes.

Preparing these requirements in advance allows you to control the scope of your abstraction layer. This means that you can spend less time

creating multiple implementations of your abstraction layer and begin using your new backward-compatible implementation sooner.

The key APIs for tabs are in `ActionBar` and `ActionBar.Tab`. These are the APIs to abstract in order to make your tabs version-aware. The requirements for this example project call for compatibility back to Eclair (API level 5) while taking advantage of the new tab features in Honeycomb (API Level 11). A diagram of the class structure to support these two implementations and their abstract base classes (or interfaces) is shown below.

Figure 1. Class diagram of abstract base classes and version-specific implementations.

Abstract ActionBar.Tab

Get started on building your tab abstraction layer by creating an abstract class representing a tab, that mirrors the `ActionBar.Tab` interface:

```
public abstract class CompatTab {
    ...
    public abstract CompatTab setText(int resId);
    public abstract CompatTab setIcon(int resId);
    public abstract CompatTab setTabListener(
            CompatTabListener callback);
    public abstract CompatTab setFragment(Fragment fragment);

    public abstract CharSequence getText();
    public abstract Drawable getIcon();
    public abstract CompatTabListener getCallback();
    public abstract Fragment getFragment();
    ...
}
```

You can use an abstract class instead of an interface here to simplify the implementation of common features such as association of tab objects with activities (not shown in the code snippet).

Abstract ActionBar Tab Methods

Next, define an abstract class that allows you to create and add tabs to an activity, like `ActionBar.newTab()` and `ActionBar.addTab()`:

```
public abstract class TabHelper {
    ...

    public CompatTab newTab(String tag) {
        // This method is implemented in a later lesson.
    }

    public abstract void addTab(CompatTab tab);

    ...
}
```

In the next lessons, you create implementations for `TabHelper` and `CompatTab` that work across both older and newer platform versions.

Next Get started Next class

Proxying to the New APIs

Previous Next Get started

This lesson teaches you to:

1. Implement Tabs Using New APIs
2. Implement CompatTabHoneycomb
3. Implement TabHelperHoneycomb

You should also read

- Action Bar
- Action Bar Tabs

Try it out

Download the sample app

TabCompat.zip

This lesson shows you how to subclass the `CompatTab` and `TabHelper` abstract classes and use new APIs. Your application can use this implementation on devices running a platform version that supports them.

Implement Tabs Using New APIs

The concrete classes for `CompatTab` and `TabHelper` that use newer APIs are a *proxy* implementation. Since the abstract classes defined in the previous lesson mirror the new APIs (class structure, method signatures, etc.), the concrete classes that use these newer APIs simply proxy

method calls and their results.

You can directly use newer APIs in these concrete classes—and not crash on earlier devices—because of lazy class loading. Classes are loaded and initialized on first access—instantiating the class or accessing one of its static fields or methods for the first time. Thus, as long as you don't instantiate the Honeycomb-specific implementations on pre-Honeycomb devices, the Dalvik VM won't throw any `VerifyError` exceptions.

A good naming convention for this implementation is to append the API level or platform version code name corresponding to the APIs required by the concrete classes. For example, the native tab implementation can be provided by `CompatTabHoneycomb` and `TabHelperHoneycomb` classes, since they rely on APIs available in Android 3.0 (API level 11) or later.

Figure 1. Class diagram for the Honeycomb implementation of tabs.

Implement CompatTabHoneycomb

`CompatTabHoneycomb` is the implementation of the `CompatTab` abstract class that `TabHelperHoneycomb` uses to reference individual tabs. `CompatTabHoneycomb` simply proxies all method calls to its contained `ActionBar.Tab` object.

Begin implementing `CompatTabHoneycomb` using the new `ActionBar.Tab` APIs:

```
public class CompatTabHoneycomb extends CompatTab {
    // The native tab object that this CompatTab acts as a proxy
for.
    ActionBar.Tab mTab;
    ...
```

```
    protected CompatTabHoneycomb(FragmentActivity activity, String
tag) {
        ...
        // Proxy to new ActionBar.newTab API
        mTab = activity.getActionBar().newTab();
    }

    public CompatTab setText(int resId) {
        // Proxy to new ActionBar.Tab.setText API
        mTab.setText(resId);
        return this;
    }

    ...
    // Do the same for other properties (icon, callback, etc.)
}
```

Implement TabHelperHoneycomb

`TabHelperHoneycomb` is the implementation of the `TabHelper` abstract class that proxies method calls to an actual `ActionBar`, obtained from its contained `Activity`.

Implement `TabHelperHoneycomb`, proxying method calls to the `ActionBar` API:

```
public class TabHelperHoneycomb extends TabHelper {
    ActionBar mActionBar;
    ...

    protected void setUp() {
        if (mActionBar == null) {
            mActionBar = mActivity.getActionBar();
            mActionBar.setNavigationMode(
                    ActionBar.NAVIGATION_MODE_TABS);
        }
    }

    public void addTab(CompatTab tab) {
        ...
        // Tab is a CompatTabHoneycomb instance, so its
        // native tab object is an ActionBar.Tab.
        mActionBar.addTab((ActionBar.Tab) tab.getTab());
    }

    // The other important method, newTab() is part of
    // the base implementation.
}
```

Next Get started Next class

Creating an Implementation with Older APIs

Previous Next Get started

This lesson teaches you to:

1. Decide on a Substitute Solution
2. Implement Tabs Using Older APIs

Try it out

Download the sample app

TabCompat.zip

This lesson discusses how to create an implementation that mirrors newer APIs yet supports older devices.

Decide on a Substitute Solution

The most challenging task in using newer UI features in a backward-compatible way is deciding on and implementing an older (fallback) solution for older platform versions. In many cases, it's possible to fulfill the purpose of these newer UI components using older UI framework features. For example:

- Action bars can be implemented using a horizontal `LinearLayout` containing image buttons, either as custom title bars or as views in your activity layout. Overflow actions can be presented under the device *Menu* button.

Action bar tabs can be implemented using a horizontal `LinearLayout` containing buttons, or using the `TabWidget` UI element.

- `NumberPicker` and `Switch` widgets can be implemented using `Spinner` and `ToggleButton` widgets, respectively.
- `ListPopupWindow` and `PopupMenu` widgets can be implemented using `PopupWindow` widgets.

There generally isn't a one-size-fits-all solution for backporting newer UI components to older devices. Be mindful of the user experience: on older devices, users may not be familiar with newer design patterns and UI components. Give some thought as to how the same functionality can be delivered using familiar elements. In many cases this is less of a concern—if newer UI components are prominent in the application ecosystem (such as the action bar), or where the interaction model is extremely simple and intuitive (such as swipe views using a `ViewPager`).

Implement Tabs Using Older APIs

To create an older implementation of action bar tabs, you can use a `TabWidget` and `TabHost` (although one can alternatively use horizontally laid-out `Button` widgets). Implement this in classes called `TabHelperEclair` and `CompatTabEclair`, since this implementation uses APIs introduced no later than Android 2.0 (Eclair).

Figure 1. Class diagram for the Eclair implementation of tabs.

The `CompatTabEclair` implementation stores tab properties such as the tab text and icon in instance variables, since there isn't an `ActionBar.Tab` object available to handle this storage:

```
public class CompatTabEclair extends CompatTab {
    // Store these properties in the instance,
    // as there is no ActionBar.Tab object.
    private CharSequence mText;
    ...

    public CompatTab setText(int resId) {
        // Our older implementation simply stores this
        // information in the object instance.
        mText = mActivity.getResources().getText(resId);
        return this;
    }

    ...
    // Do the same for other properties (icon, callback, etc.)
}
```

The `TabHelperEclair` implementation makes use of methods on the `TabHost` widget for creating `TabHost.TabSpec` objects and tab indicators:

```
public class TabHelperEclair extends TabHelper {
    private TabHost mTabHost;
    ...

    protected void setUp() {
        if (mTabHost == null) {
            // Our activity layout for pre-Honeycomb devices
            // must contain a TabHost.
            mTabHost = (TabHost) mActivity.findViewById(
                    android.R.id.tabhost);
            mTabHost.setup();
        }
    }

    public void addTab(CompatTab tab) {
        ...
        TabSpec spec = mTabHost
                .newTabSpec(tag)
                .setIndicator(tab.getText()); // And optional icon
        ...
        mTabHost.addTab(spec);
    }

    // The other important method, newTab() is part of
    // the base implementation.
}
```

You now have two implementations of `CompatTab` and `TabHelper`: one that works on devices running Android 3.0 or later and uses new APIs, and another that works on devices running Android 2.0 or later and uses older APIs. The next lesson discusses using these implementations in your application.

Next Get started Next class

Using the Version-Aware Component

Previous Next Get started

This lesson teaches you to:

1. Add the Switching Logic
2. Create a Version-Aware Activity Layout
3. Use TabHelper in Your Activity

Try it out

Download the sample app

TabCompat.zip

Now that you have two implementations of `TabHelper` and `CompatTab`—one for Android 3.0 and later and one for earlier versions of the platform—it's time to do something with these implementations. This lesson discusses creating the logic for switching between these implementations, creating version-aware layouts, and finally using the backward-compatible UI component.

Add the Switching Logic

The `TabHelper` abstract class acts as a factory for creating version-appropriate `TabHelper` and `CompatTab` instances, based on the current device's platform version:

```
public abstract class TabHelper {
    ...
    // Usage is TabHelper.createInstance(activity)
    public static TabHelper createInstance(FragmentActivity activity)
```

```
{
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
            return new TabHelperHoneycomb(activity);
        } else {
            return new TabHelperEclair(activity);
        }
    }

    // Usage is mTabHelper.newTab("tag")
    public CompatTab newTab(String tag) {
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
            return new CompatTabHoneycomb(mActivity, tag);
        } else {
            return new CompatTabEclair(mActivity, tag);
        }
    }
    ...
}
```

Create a Version-Aware Activity Layout

The next step is to provide layouts for your activity that can support the two tab implementations. For the older implementation (`TabHelperEclair`), you need to ensure that your activity layout contains a `TabWidget` and `TabHost`, along with a container for tab contents:

res/layout/main.xml:

```
<!-- This layout is for API level 5-10 only. -->
<TabHost xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@android:id/tabhost"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <LinearLayout
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:padding="5dp">

        <TabWidget
            android:id="@android:id/tabs"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />

        <FrameLayout
            android:id="@android:id/tabcontent"
            android:layout_width="match_parent"
            android:layout_height="0dp"
            android:layout_weight="1" />

    </LinearLayout>
</TabHost>
```

For the `TabHelperHoneycomb` implementation, all you need is a `FrameLayout` to contain the tab contents, since the tab indicators are provided by the `ActionBar`:

res/layout-v11/main.xml:

```
<FrameLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@android:id/tabcontent"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

At runtime, Android will decide which version of the `main.xml` layout to inflate depending on the platform version. This is the same logic shown in the previous section to determine which `TabHelper` implementation to use.

Use TabHelper in Your Activity

In your activity's `onCreate()` method, you can obtain a `TabHelper` object and add tabs with the following code:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    setContentView(R.layout.main);

    TabHelper tabHelper = TabHelper.createInstance(this);
    tabHelper.setUp();

    CompatTab photosTab = tabHelper
            .newTab("photos")
            .setText(R.string.tab_photos);
    tabHelper.addTab(photosTab);

    CompatTab videosTab = tabHelper
            .newTab("videos")
            .setText(R.string.tab_videos);
    tabHelper.addTab(videosTab);
}
```

When running the application, this code inflates the correct activity layout and instantiates either a `TabHelperHoneycomb` or `TabHelperEclair` object. The concrete class that's actually used is opaque to the activity, since they share the common `TabHelper` interface.

Below are two screenshots of this implementation running on an Android 2.3 and Android 4.0 device.

Figure 1. Example screenshots of backward-compatible tabs running on an Android 2.3 device (using `TabHelperEclair`) and an Android 4.0 device (using `TabHelperHoneycomb`).

Next Get started Next class

Implementing Accessibility

Previous Next Get started

Dependencies and prerequisites

- Android 2.0 (API Level 5) or higher

You should also read

- Accessibility

When it comes to reaching as wide a userbase as possible, it's important to pay attention to accessibility in your Android application. Cues in your user interface that may work for a majority of users, such as a visible change in state when a button is pressed, can be less optimal if the user is visually impaired.

This class shows you how to make the most of the accessibility features built into the Android framework. It covers how to optimize your app for accessibility, leveraging platform features like focus navigation and content descriptions. It also covers how to build accessibility services, that can facilitate user interaction with **any** Android application, not just your own.

Lessons

Developing Accessible Applications

Learn to make your Android application accessible. Allow for easy navigation with a keyboard or directional pad, set labels and fire events that can be interpreted by an accessibility service to facilitate a smooth user experience.

Developing Accessibility Services

Develop an accessibility service that listens for accessibility events, mines those events for information like event type and content descriptions, and uses that information to communicate with the user. The example will use a text-to-speech engine to speak to the user.

Next Get started Next class

Developing Accessible Applications

Previous Next Get started

This lesson teaches you to

1. Add Content Descriptions
2. Design for Focus Navigation
3. Fire Accessibility Events
4. Test Your Application

You should also read

- Making Applications Accessible

Android has several accessibility-focused features baked into the platform, which make it easy to optimize your application for those with visual or physical disabilities. However, it's not always obvious what the correct optimizations are, or the easiest way to leverage the framework toward this purpose. This lesson shows you how to implement the strategies and platform features that make for a great accessibility-enabled Android application.

Add Content Descriptions

A well-designed user interface (UI) often has elements that don't require an explicit label to indicate their purpose to the user. A checkbox next to an item in a task list application has a fairly obvious purpose, as does a trash can in a file manager application. However, to your users with vision impairment, other UI cues are needed.

Fortunately, it's easy to add labels to UI elements in your application that can be read out loud to your user by a speech-based accessibility service like TalkBack . If you have a label that's likely not to change during the lifecycle of the application (such as "Pause" or "Purchase"), you can add it via the XML layout, by setting a UI element's `android:contentDescription` attribute, like in this example:

```
<Button
    android:id="@+id/pause_button"
    android:src="@drawable/pause"
    android:contentDescription="@string/pause"/>
```

However, there are plenty of situations where it's desirable to base the content description on some context, such as the state of a toggle button, or a piece selectable data like a list item. To edit the content description at runtime, use the `setContentDescription()` method, like this:

```
String contentDescription = "Select " + strValues[position];
label.setContentDescription(contentDescription);
```

This addition to your code is the simplest accessibility improvement you can make to your application, but one of the most useful. Try to add content descriptions wherever there's useful information, but avoid the web-developer pitfall of labelling *everything* with useless information. For instance, don't set an application icon's content description to "app icon". That just increases the noise a user needs to navigate in order to pull useful information from your interface.

Try it out! Download TalkBack (an accessibility service published by Google) and enable it in **Settings > Accessibility > TalkBack**. Then navigate around your own application and listen for the audible cues provided by TalkBack.

Design for Focus Navigation

Your application should support more methods of navigation than the touch screen alone. Many Android devices come with navigation hardware other than the touchscreen, like a D-Pad, arrow keys, or a trackball. In addition, later Android releases also support connecting

external devices like keyboards via USB or bluetooth.

In order to enable this form of navigation, all navigational elements that the user should be able to navigate to need to be set as focusable. This modification can be done at runtime using the `View.setFocusable()` method on that UI control, or by setting the `android:focusable` attrubute in your XML layout files.

Also, each UI control has 4 attributes, `android:nextFocusUp`, `android:nextFocusDown`, `android:nextFocusLeft`, and `android:nextFocusRight`, which you can use to designate the next view to receive focus when the user navigates in that direction. While the platform determines navigation sequences automatically based on layout proximity, you can use these attributes to override that sequence if it isn't appropriate in your application.

For instance, here's how you represent a button and label, both focusable, such that pressing down takes you from the button to the text view, and pressing up would take you back to the button.

```
<Button android:id="@+id/doSomething"
    android:focusable="true"
    android:nextFocusDown=”@id/label”
    ... />
<TextView android:id="@+id/label"
    android:focusable=”true”
    android:text="@string/labelText"
    android:nextFocusUp=”@id/doSomething”
    ... />
```

Verify that your application works intuitively in these situations. The easiest way is to simply run your application in the Android emulator, and navigate around the UI with the emulator's arrow keys, using the OK button as a replacement for touch to select UI controls.

Fire Accessibility Events

If you're using the view components in the Android framework, an `AccessibilityEvent` is created whenever you select an item or change focus in your UI. These events are examined by the accessibility service, enabling it to provide features like text-to-speech to the user.

If you write a custom view, make sure it fires events at the appropriate times. Generate events by calling `sendAccessibilityEvent(int)`, with a parameter representing the type of event that occurred. A complete list of the event types currently supported can be found in the `AccessibilityEvent` reference documentation.

As an example, if you want to extend an image view such that you can write captions by typing on the keyboard when it has focus, it makes sense to fire an `TYPE_VIEW_TEXT_CHANGED` event, even though that's not normally built into image views. The code to generate that event would look like this:

```
public void onTextChanged(String before, String after) {
    ...
    if (AccessibilityManager.getInstance(mContext).isEnabled()) {
sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_TEXT_CHANGED);
    }
    ...
}
```

Test Your Application

Be sure to test the accessibility functionality as you add it to your application. In order to test the content descriptions and Accessibility events, install and enable an accessibility service. One option is Talkback , a free, open source screen reader available on Google Play. With the service enabled, test all the navigation flows through your application and listen to the spoken feedback.

Also, attempt to navigate your application using a directional controller, instead of the touch screen. You can use a physical device with a d-pad or trackball if one is available. If not, use the Android emulator and it's simulated keyboard controls.

Between the service providing feedback and the directional navigation through your application, you should get a sense of what your application is like to navigate without any visual cues. Fix problem areas as they appear, and you'll end up with with a more accessible Android application.

Next Get started Next class

Developing an Accessibility Service

Previous Next Get started

This lesson teaches you to

1. Create Your Accessibility Service
2. Configure Your Accessibility Service
3. Respond to AccessibilityEvents
4. Query the View Heirarchy for More Context

You should also read

- Building Accessibility Services

Accessibility services are a feature of the Android framework designed to provide alternative navigation feedback to the user on behalf of applications installed on Android devices. An accessibility service can communicate to the user on the application's behalf, such as converting text to speech, or haptic feedback when a user is hovering on an important area of the screen. This lesson covers how to create an accessibility service, process information received from the application, and report that information back to the user.

Create Your Accessibility Service

An accessibility service can be bundled with a normal application, or created as a standalone Android project. The steps to creating the service are the same in either situation. Within your project, create a class that extends `AccessibilityService`.

```
package com.example.android.apis.accessibility;

import android.accessibilityservice.AccessibilityService;

public class MyAccessibilityService extends AccessibilityService {
...
    @Override
    public void onAccessibilityEvent(AccessibilityEvent event) {
    }

    @Override
    public void onInterrupt() {
    }

...
}
```

Like any other service, you also declare it in the manifest file. Remember to specify that it handles the `android.accessibilityservice` intent, so that the service is called when applications fire an `AccessibilityEvent`.

```
<application ...>
...
<service android:name=".MyAccessibilityService">
     <intent-filter>
         <action
android:name="android.accessibilityservice.AccessibilityService" />
     </intent-filter>
     . . .
</service>
...
</application>
```

If you created a new project for this service, and don't plan on having an application, you can remove the starter Activity class (usually called MainActivity.java) from your source. Remember to also remove the corresponding activity element from your manifest.

Configure Your Accessibility Service

Setting the configuration variables for your accessibility service tells the system how and when you want it to run. Which event types would you like to respond to? Should the service be active for all applications, or only specific package names? What different feedback types does it use?

You have two options for how to set these variables. The backwards-compatible option is to set them in code, using `setServiceInfo(android.accessibilityservice.AccessibilityServiceInfo)`

. To do that, override the `onServiceConnected()` method and configure your service in there.

```
@Override
public void onServiceConnected() {
    // Set the type of events that this service wants to listen to.
Others
    // won't be passed to this service.
    info.eventTypes = AccessibilityEvent.TYPE_VIEW_CLICKED |
            AccessibilityEvent.TYPE_VIEW_FOCUSED;

    // If you only want this service to work with specific
applications, set their
    // package names here.  Otherwise, when the service is
activated, it will listen
    // to events from all applications.
    info.packageNames = new String[]
            {"com.example.android.myFirstApp",
"com.example.android.mySecondApp"};

    // Set the type of feedback your service will provide.
    info.feedbackType = AccessibilityServiceInfo.FEEDBACK_SPOKEN;

    // Default services are invoked only if no package-specific ones
are present
    // for the type of AccessibilityEvent generated.  This service
*is*
    // application-specific, so the flag isn't necessary.  If this
was a
    // general-purpose service, it would be worth considering
setting the
    // DEFAULT flag.

    // info.flags = AccessibilityServiceInfo.DEFAULT;

    info.notificationTimeout = 100;

    this.setServiceInfo(info);

}
```

Starting with Android 4.0, there is a second option available: configure the service using an XML file. Certain configuration options like `canRetrieveWindowContent` are only available if you configure your service using XML. The same configuration options above, defined using XML, would look like this:

```
<accessibility-service

android:accessibilityEventTypes="typeViewClicked|typeViewFocused"
     android:packageNames="com.example.android.myFirstApp,
com.example.android.mySecondApp"
     android:accessibilityFeedbackType="feedbackSpoken"
     android:notificationTimeout="100"
```

```
android:settingsActivity="com.example.android.apis.accessibility.TestBa

     android:canRetrieveWindowContent="true"
/>
```

If you go the XML route, be sure to reference it in your manifest, by adding a <meta-data> tag to your service declaration, pointing at the XML file. If you stored your XML file in `res/xml/serviceconfig.xml`, the new tag would look like this:

```
<service android:name=".MyAccessibilityService">
     <intent-filter>
         <action
android:name="android.accessibilityservice.AccessibilityService" />
     </intent-filter>
     <meta-data android:name="android.accessibilityservice"
     android:resource="@xml/serviceconfig" />
</service>
```

Respond to AccessibilityEvents

Now that your service is set up to run and listen for events, write some code so it knows what to do when an `AccessibilityEvent` actually arrives! Start by overriding the `onAccessibilityEvent(AccessibilityEvent)` method. In that method, use `getEventType()` to determine the type of event, and `getContentDescription()` to extract any label text associated with the view that fired the event.

```
@Override
public void onAccessibilityEvent(AccessibilityEvent event) {
    final int eventType = event.getEventType();
    String eventText = null;
    switch(eventType) {
        case AccessibilityEvent.TYPE_VIEW_CLICKED:
            eventText = "Focused: ";
            break;
        case AccessibilityEvent.TYPE_VIEW_FOCUSED:
            eventText = "Focused: ";
            break;
    }

    eventText = eventText + event.getContentDescription();

    // Do something nifty with this text, like speak the composed
string
    // back to the user.
    speakToUser(eventText);
    ...
}
```

Query the View Heirarchy for More Context

This step is optional, but highly useful. One of the new features in Android 4.0 (API Level 14) is the ability for an `AccessibilityService` to query the view hierarchy, collecting information about the UI component that generated an event, and its parent and children. In order to do this, make sure that you set the following line in your XML configuration:

```
android:canRetrieveWindowContent="true"
```

Once that's done, get an `AccessibilityNodeInfo` object using `getSource()`. This call only returns an object if the window where the event originated is still the active window. If not, it will return null, so *behave accordingly*. The following example is a snippet of code that, when it receives an event, does the following:

1. Immediately grab the parent of the view where the event originated
2. In that view, look for a label and a check box as children views
3. If it finds them, create a string to report to the user, indicating the label and whether it was checked or not.
4. If at any point a null value is returned while traversing the view hierarchy, the method quietly gives up.

```
// Alternative onAccessibilityEvent, that uses AccessibilityNodeInfo

@Override
public void onAccessibilityEvent(AccessibilityEvent event) {

    AccessibilityNodeInfo source = event.getSource();
    if (source == null) {
        return;
    }

    // Grab the parent of the view that fired the event.
    AccessibilityNodeInfo rowNode = getListItemNodeInfo(source);
    if (rowNode == null) {
        return;
    }

    // Using this parent, get references to both child nodes, the
label and the checkbox.
    AccessibilityNodeInfo labelNode = rowNode.getChild(0);
    if (labelNode == null) {
        rowNode.recycle();
        return;
    }
```

```
    AccessibilityNodeInfo completeNode = rowNode.getChild(1);
    if (completeNode == null) {
        rowNode.recycle();
        return;
    }

    // Determine what the task is and whether or not it's complete,
based on
    // the text inside the label, and the state of the check-box.
    if (rowNode.getChildCount() < 2 ||
!rowNode.getChild(1).isCheckable()) {
        rowNode.recycle();
        return;
    }

    CharSequence taskLabel = labelNode.getText();
    final boolean isComplete = completeNode.isChecked();
    String completeStr = null;

    if (isComplete) {
        completeStr = getString(R.string.checked);
    } else {
        completeStr = getString(R.string.not_checked);
    }
    String reportStr = taskLabel + completeStr;
    speakToUser(reportStr);
}
```

Now you have a complete, functioning accessibility service. Try configuring how it interacts with the user, by adding Android's text-to-speech engine, or using a `Vibrator` to provide haptic feedback!

Next Get started Next class

Managing the System UI

Previous Next Get started

Dependencies and prerequisites

- Android 1.6 (API Level 4) or higher

You should also read

- Action Bar API Guide
- Android Design Guide

Try it out

Get the sample

ImmersiveMode sample

Design Guide

System Bars

Video

DevBytes: Android 4.4 Immersive Mode

Figure 1. System bars, including the [1] status bar, and [2] navigation bar.

The system bars are screen areas dedicated to the display of notifications, communication of device status, and device navigation. Typically the system bars (which consist of the status and navigation bars, as shown in figure 1) are displayed concurrently with your app. Apps that display immersive content, such as movies or images, can temporarily dim the system bar icons for a less distracting experience, or temporarily hide the bars for a fully immersive experience.

If you're familiar with the Android Design Guide, you know the importance of designing your apps to conform to standard Android UI guidelines and usage patterns. You should carefully consider your users' needs and expectations before modifying the system bars, since they give users a standard way of navigating a device and viewing its status.

This class describes how to dim or hide system bars across different versions of Android to create an immersive user experience, while still preserving easy access to the system bars.

Lessons

Dimming the System Bars
Learn how to dim the status and navigation bars.

Hiding the Status Bar
Learn how to hide the status bar on different versions of Android.

Hiding the Navigation Bar
Learn how to hide the navigation bar, in addition to the status bar.

Using Immersive Full-Screen Mode
Learn how to create a fully immersive experience in your app.

Responding to UI Visibility Changes
Learn how to register a listener to get notified of system UI visibility changes so that you can adjust your app's UI accordingly.

Next Get started Next class

Dimming the System Bars

Previous Next Get started

This lesson teaches you to

1. Dim the Status and Navigation Bars
2. Reveal the Status and Navigation Bars

You should also read

- Action Bar API Guide
- Android Design Guide

Try it out

Get the sample

ImmersiveMode sample

This lesson describes how to dim the system bars (that is, the status and the navigation bars) on Android 4.0 (API level 14) and higher. Android does not provide a built-in way to dim the system bars on earlier versions.

When you use this approach, the content doesn't resize, but the icons in the system bars visually recede. As soon as the user touches either the status bar or the navigation bar area of the screen, both bars become fully visible. The advantage of this approach is that the bars are still present but their details are obscured, thus creating an immersive experience without sacrificing easy access to the bars.

Dim the Status and Navigation Bars

You can dim the status and notification bars on Android 4.0 and higher using the `SYSTEM_UI_FLAG_LOW_PROFILE` flag, as follows:

```
// This example uses decor view, but you can use any visible view.
View decorView = getActivity().getWindow().getDecorView();
int uiOptions = View.SYSTEM_UI_FLAG_LOW_PROFILE;
decorView.setSystemUiVisibility(uiOptions);
```

As soon as the user touches the status or navigation bar, the flag is cleared, causing the bars to be undimmed. Once the flag has been cleared, your app needs to reset it if you want to dim the bars again.

Figure 1 shows a gallery image in which the navigation bar is dimmed (note that the Gallery app completely hides the status bar; it doesn't dim it). Notice that the navigation bar (right side of the image) has faint white dots on it to represent the navigation controls:

Figure 1. Dimmed system bars.

Figure 2 shows the same gallery image, but with the system bars displayed:

Figure 2. Visible system bars.

Reveal the Status and Navigation Bars

If you want to programmatically clear flags set with `setSystemUiVisibility()`, you can do so as follows:

```
View decorView = getActivity().getWindow().getDecorView();
// Calling setSystemUiVisibility() with a value of 0 clears
// all flags.
decorView.setSystemUiVisibility(0);
```

Next Get started Next class

Hiding the Status Bar

Previous Next Get started

This lesson teaches you to

1. Hide the Status Bar on Android 4.0 and Lower
2. Hide the Status Bar on Android 4.1 and Higher
3. Hide the Status Bar on Android 4.4 and Higher
4. Make Content Appear Behind the Status Bar
5. Synchronize the Status Bar with Action Bar Transition

You should also read

- Action Bar API Guide
- Android Design Guide

Try it out

Get the sample

ImmersiveMode sample

This lesson describes how to hide the status bar on different versions of Android. Hiding the status bar (and optionally, the navigation bar) lets the content use more of the display space, thereby providing a more immersive user experience.

Figure 1 shows an app with a visible status bar:

Figure 1. Visible status bar.

Figure 2 shows an app with a hidden status bar. Note that the action bar is hidden too. You should never show the action bar without the status bar.

Figure 2. Hidden status bar.

Hide the Status Bar on Android 4.0 and Lower

You can hide the status bar on Android 4.0 (API level 14) and lower by setting `WindowManager` flags. You can do this programmatically or by setting an activity theme in your app's manifest file. Setting an activity theme in your app's manifest file is the preferred approach if the status

bar should always remain hidden in your app (though strictly speaking, you could programmatically override the theme if you wanted to). For example:

```
<application
    ...
    android:theme="@android:style/Theme.Holo.NoActionBar.Fullscreen"
>
    ...
</application>
```

The advantages of using an activity theme are as follows:

- It's easier to maintain and less error-prone than setting a flag programmatically.
- It results in smoother UI transitions, because the system has the information it needs to render your UI before instantiating your app's main activity.

Alternatively, you can programmatically set `WindowManager` flags. This approach makes it easier to hide and show the status bar as the user interacts with your app:

```
public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // If the Android version is lower than Jellybean, use this
call to hide
        // the status bar.
        if (Build.VERSION.SDK_INT < 16) {

getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
                     WindowManager.LayoutParams.FLAG_FULLSCREEN);
        }
        setContentView(R.layout.activity_main);
    }
    ...
}
```

When you set `WindowManager` flags (whether through an activity theme or programmatically), the flags remain in effect unless your app clears them.

You can use `FLAG_LAYOUT_IN_SCREEN` to set your activity layout to use the same screen area that's available when you've enabled `FLAG_FULLSCREEN`. This prevents your content from resizing when the status bar hides and

shows.

Hide the Status Bar on Android 4.1 and Higher

You can hide the status bar on Android 4.1 (API level 16) and higher by using `setSystemUiVisibility()`. `setSystemUiVisibility()` sets UI flags at the individual view level; these settings are aggregated to the window level. Using `setSystemUiVisibility()` to set UI flags gives you more granular control over the system bars than using `WindowManager` flags. This snippet hides the status bar:

```
View decorView = getWindow().getDecorView();
// Hide the status bar.
int uiOptions = View.SYSTEM_UI_FLAG_FULLSCREEN;
decorView.setSystemUiVisibility(uiOptions);
// Remember that you should never show the action bar if the
// status bar is hidden, so hide that too if necessary.
ActionBar actionBar = getActionBar();
actionBar.hide();
```

Note the following:

- Once UI flags have been cleared (for example, by navigating away from the activity), your app needs to reset them if you want to hide the bars again. See Responding to UI Visibility Changes for a discussion of how to listen for UI visibility changes so that your app can respond accordingly.
- Where you set the UI flags makes a difference. If you hide the system bars in your activity's `onCreate()` method and the user presses Home, the system bars will reappear. When the user reopens the activity, `onCreate()` won't get called, so the system bars will remain visible. If you want system UI changes to persist as the user navigates in and out of your activity, set UI flags in `onResume()` or `onWindowFocusChanged()`.
- The method `setSystemUiVisibility()` only has an effect if the view you call it from is visible.
- Navigating away from the view causes flags set with `setSystemUiVisibility()` to be cleared.

Make Content Appear Behind the Status Bar

On Android 4.1 and higher, you can set your application's content to appear behind the status bar, so that the content doesn't resize as the status bar hides and shows. To do this, use `SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN`. You may also need to use `SYSTEM_UI_FLAG_LAYOUT_STABLE` to help your app maintain a stable layout.

When you use this approach, it becomes your responsibility to ensure that critical parts of your app's UI (for example, the built-in controls in a Maps application) don't end up getting covered by system bars. This could make your app unusable. In most cases you can handle this by adding the `android:fitsSystemWindows` attribute to your XML layout file, set to `true`. This adjusts the padding of the parent `ViewGroup` to leave space for the system windows. This is sufficient for most applications.

In some cases, however, you may need to modify the default padding to get the desired layout for your app. To directly manipulate how your content lays out relative to the system bars (which occupy a space known as the window's "content insets"), override `fitSystemWindows(Rect insets)`. The `fitSystemWindows()` method is called by the view hierarchy when the content insets for a window have changed, to allow the window to adjust its content accordingly. By overriding this method you can handle the insets (and hence your app's layout) however you want.

Synchronize the Status Bar with Action Bar Transition

On Android 4.1 and higher, to avoid resizing your layout when the action bar hides and shows, you can enable overlay mode for the action bar. When in overlay mode, your activity layout uses all the space available as if the action bar is not there and the system draws the action bar in front of your layout. This obscures some of the layout at the top, but now when the action bar hides or appears, the system does not need to resize your layout and the transition is seamless.

To enable overlay mode for the action bar, you need to create a custom theme that extends an existing theme with an action bar and set the `android:windowActionBarOverlay` attribute to `true`. For more discussion of this topic, see Overlaying the Action Bar in the Adding the Action Bar class.

Then use `SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN`, as described above, to set your activity layout to use the same screen area that's available when you've enabled `SYSTEM_UI_FLAG_FULLSCREEN`. When you want to hide the system UI, use `SYSTEM_UI_FLAG_FULLSCREEN`. This also hides the action bar (because `windowActionBarOverlay="true")` and does so with a coordinated animation when both hiding and showing the two.

Next Get started Next class

Hiding the Navigation Bar

Previous Next Get started

This lesson teaches you to

1. Hide the Navigation Bar on 4.0 and Higher
2. Make Content Appear Behind the Navigation Bar

You should also read

- Action Bar API Guide
- Android Design Guide

Try it out

Get the sample

ImmersiveMode sample

This lesson describes how to hide the navigation bar, which was introduced in Android 4.0 (API level 14).

Even though this lesson focuses on hiding the navigation bar, you should design your app to hide the status bar at the same time, as described in Hiding the Status Bar. Hiding the navigation and status bars (while still keeping them readily accessible) lets the content use the entire display space, thereby providing a more immersive user experience.

Figure 1. Navigation bar.

Hide the Navigation Bar on 4.0 and Higher

You can hide the navigation bar on Android 4.0 and higher using the `SYSTEM_UI_FLAG_HIDE_NAVIGATION` flag. This snippet hides both the navigation bar and the status bar:

```
View decorView = getWindow().getDecorView();
// Hide both the navigation bar and the status bar.
// SYSTEM_UI_FLAG_FULLSCREEN is only available on Android 4.1 and
higher, but as
// a general rule, you should design your app to hide the status bar
whenever you
// hide the navigation bar.
int uiOptions = View.SYSTEM_UI_FLAG_HIDE_NAVIGATION
              | View.SYSTEM_UI_FLAG_FULLSCREEN;
decorView.setSystemUiVisibility(uiOptions);
```

Note the following:

- With this approach, touching anywhere on the screen causes the navigation bar (and status bar) to reappear and remain visible. The user interaction causes the flags to be be cleared.
- Once the flags have been cleared, your app needs to reset them if you want to hide the bars again. See Responding to UI Visibility Changes for a discussion of how to listen for UI visibility changes so that your app can respond accordingly.
- Where you set the UI flags makes a difference. If you hide the system bars in your activity's `onCreate()` method and the user presses Home, the system bars will reappear. When the user reopens the activity, `onCreate()` won't get called, so the system bars

will remain visible. If you want system UI changes to persist as the user navigates in and out of your activity, set UI flags in `onResume()` or `onWindowFocusChanged()`.
- The method `setSystemUiVisibility()` only has an effect if the view you call it from is visible.
- Navigating away from the view causes flags set with `setSystemUiVisibility()` to be cleared.

Make Content Appear Behind the Navigation Bar

On Android 4.1 and higher, you can set your application's content to appear behind the navigation bar, so that the content doesn't resize as the navigation bar hides and shows. To do this, use `SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION`. You may also need to use `SYSTEM_UI_FLAG_LAYOUT_STABLE` to help your app maintain a stable layout.

When you use this approach, it becomes your responsibility to ensure that critical parts of your app's UI don't end up getting covered by system bars. For more discussion of this topic, see the Hiding the Status Bar lesson.

Next Get started Next class

Using Immersive Full-Screen Mode

Previous Next Get started

This lesson teaches you to

1. Choose an Approach
2. Use Non-Sticky Immersion
3. Use Sticky Immersion

You should also read

- Action Bar API Guide
- Android Design Guide

Try it out

Get the sample

ImmersiveMode sample

Video

DevBytes: Android 4.4 Immersive Mode

Android 4.4 (API Level 19) introduces a new `SYSTEM_UI_FLAG_IMMERSIVE` flag for `setSystemUiVisibility()` that lets your app go truly "full screen." This flag, when combined with the `SYSTEM_UI_FLAG_HIDE_NAVIGATION` and `SYSTEM_UI_FLAG_FULLSCREEN` flags, hides the navigation and status bars and lets your app capture all touch events on the screen.

When immersive full-screen mode is enabled, your activity continues to receive all touch events. The user can reveal the system bars with an inward swipe along the region where the system bars normally appear. This clears the `SYSTEM_UI_FLAG_HIDE_NAVIGATION` flag (and the `SYSTEM_UI_FLAG_FULLSCREEN` flag, if applied) so the system bars become visible. This also triggers your `View.OnSystemUiVisibilityChangeListener`, if set. However, if you'd like the

system bars to automatically hide again after a few moments, you can instead use the `SYSTEM_UI_FLAG_IMMERSIVE_STICKY` flag. Note that the "sticky" version of the flag doesn't trigger any listeners, as system bars temporarily shown in this mode are in a transient state.

Figure 1 illustrates the different "immersive mode" states:

Figure 1. Immersive mode states.

In figure 1:

1. **Non-immersive mode**—This is how the app appears before it enters immersive mode. It is also how the app appears if you use the `IMMERSIVE` flag, and the user swipes to display the system bars, thereby clearing the `SYSTEM_UI_FLAG_HIDE_NAVIGATION` and `SYSTEM_UI_FLAG_FULLSCREEN` flags. Once these flags are cleared, the system bars reappear and remain visible.

   Note that it's best practice to keep all UI controls in sync with the system bars, to minimize the number of states your screen can be in. This provides a more seamless user experience. So here all UI controls are displayed along with the status bars. Once the app enters immersive mode, the UI controls are hidden along with the system bars. To ensure that your UI visibility stays in sync with system bar visibility, make sure to provide an appropriate `View.OnSystemUiVisibilityChangeListener` to watch for changes, as described in Responding to UI Visibility Changes.

2. **Reminder bubble**—The system displays a reminder bubble the first time users enter immersive mode in your app. The reminder bubble reminds users how to display the system bars.

   Note: If you want to force the reminder bubble to appear for testing purposes, you can do so by putting the app in immersive mode, turning off the screen with the power button, and then turning the screen back on again within 5 seconds.

3. **Immersive mode**—This is the app in immersive mode, with the system bars and other UI controls hidden. You can achieve this state with either `IMMERSIVE` or `IMMERSIVE_STICKY`.
4. **Sticky flag**—This is the UI you see if you use the `IMMERSIVE_STICKY` flag, and the user swipes to display the system bars. Semi-transparent bars temporarily appear and then hide again. The act of swiping doesn't clear any flags, nor does it trigger your system UI visibility change listeners, because the transient appearance of the system bars isn't considered a UI visibility change.

Note: Remember that the "immersive" flags only take effect if you use them in conjunction with `SYSTEM_UI_FLAG_HIDE_NAVIGATION`, `SYSTEM_UI_FLAG_FULLSCREEN`, or both. You can just use one or the other, but it's common to hide both the status and the navigation bar when you're implementing "full immersion" mode.

Choose an Approach

The flags `SYSTEM_UI_FLAG_IMMERSIVE` and `SYSTEM_UI_FLAG_IMMERSIVE_STICKY` both provide an immersive experience, but with the differences in behavior described above. Here are examples of when you would use one flag vs. the other:

- If you're building a book reader, news reader, or a magazine, use the `IMMERSIVE` flag in conjunction with `SYSTEM_UI_FLAG_FULLSCREEN` and `SYSTEM_UI_FLAG_HIDE_NAVIGATION`. Because users may want to access the action bar and other UI controls somewhat frequently, but not be bothered with any UI elements while flipping through content, `IMMERSIVE` is a good option for this use case.
- If you're building a truly immersive app, where you expect users to interact near the edges of the screen and you don't expect them to need frequent access to the system UI, use the `IMMERSIVE_STICKY` flag in conjunction with `SYSTEM_UI_FLAG_FULLSCREEN` and `SYSTEM_UI_FLAG_HIDE_NAVIGATION`. For example, this approach might be suitable for a game or a drawing app.

- If you're building a video player or some other app that requires minimal user interaction, you can probably get by with the lean back approach, available since Android 4.0 (API Level 14). For this type of app, simply using `SYSTEM_UI_FLAG_FULLSCREEN` and `SYSTEM_UI_FLAG_HIDE_NAVIGATION` should be sufficient. Don't use the "immersive" flags in this case.

Use Non-Sticky Immersion

When you use the `SYSTEM_UI_FLAG_IMMERSIVE` flag, it hides the system bars based on what other UI flags you have set (`SYSTEM_UI_FLAG_HIDE_NAVIGATION`, `SYSTEM_UI_FLAG_FULLSCREEN`, or both). When the user swipes inward in a system bars region, the system bars reappear and remain visible.

It's good practice to include other system UI flags (such as `SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION` and `SYSTEM_UI_FLAG_LAYOUT_STABLE`) to keep the content from resizing when the system bars hide and show. You should also make sure that the action bar and other UI controls are hidden at the same time. This snippet demonstrates how to hide and show the status and navigation bars, without resizing the content:

```
// This snippet hides the system bars.
private void hideSystemUI() {
    // Set the IMMERSIVE flag.
    // Set the content to appear under the system bars so that the content
    // doesn't resize when the system bars hide and show.
    mDecorView.setSystemUiVisibility(
            View.SYSTEM_UI_FLAG_LAYOUT_STABLE
            | View.SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION
            | View.SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN
            | View.SYSTEM_UI_FLAG_HIDE_NAVIGATION // hide nav bar
            | View.SYSTEM_UI_FLAG_FULLSCREEN // hide status bar
            | View.SYSTEM_UI_FLAG_IMMERSIVE);
}

// This snippet shows the system bars. It does this by removing all the
flags
// except for the ones that make the content appear under the system bars.
private void showSystemUI() {
    mDecorView.setSystemUiVisibility(
            View.SYSTEM_UI_FLAG_LAYOUT_STABLE
            | View.SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION
            | View.SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN);
}
```

You may also want to implement the following in conjunction with the `IMMERSIVE` flag to provide a better user experience:

- Register a listener so that your app can get notified of system UI visibility changes, as described in Responding to UI Visibility Changes.
- Implement `onWindowFocusChanged()`. If you gain window focus, you may

want to re-hide the system bars. If you lose window focus, for example due to a dialog or pop up menu showing above your app, you'll probably want to cancel any pending "hide" operations you previously scheduled with `Handler.postDelayed()` or something similar.

- Implement a `GestureDetector` that detects `onSingleTapUp(MotionEvent)`, to allow users to manually toggle the visibility of the system bars by touching your content. Simple click listeners aren't the best solution for this because they get triggered even if the user drags a finger across the screen (assuming the click target takes up the whole screen).

For more discussion of these topics, watch the video DevBytes: Android 4.4 Immersive Mode.

Use Sticky Immersion

When you use the `SYSTEM_UI_FLAG_IMMERSIVE_STICKY` flag, an inward swipe in the system bars areas causes the bars to temporarily appear in a semi-transparent state, but no flags are cleared, and your system UI visibility change listeners are not triggered. The bars automatically hide again after a short delay, or if the user interacts with the middle of the screen.

Figure 2 shows the semi-transparent system bars that briefly appear and then hide again when you use the `IMMERSIVE_STICKY` flag.

Figure 2. Auto-hiding system bars.

Below is a simple approach to using this flag. Any time the window receives focus, simply set the `IMMERSIVE_STICKY` flag, along with the other flags discussed in Use IMMERSIVE. For example:

```
@Override
public void onWindowFocusChanged(boolean hasFocus) {
        super.onWindowFocusChanged(hasFocus);
    if (hasFocus) {
        decorView.setSystemUiVisibility(
                View.SYSTEM_UI_FLAG_LAYOUT_STABLE
                | View.SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION
                | View.SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN
                | View.SYSTEM_UI_FLAG_HIDE_NAVIGATION
                | View.SYSTEM_UI_FLAG_FULLSCREEN
                | View.SYSTEM_UI_FLAG_IMMERSIVE_STICKY);}
}
```

Note: If you like the auto-hiding behavior of `IMMERSIVE_STICKY` but need to show your own UI controls as well, just use `IMMERSIVE` combined with `Handler.postDelayed()` or something similar to re-enter immersive mode after a few seconds.

Next Get started Next class

Responding to UI Visibility Changes

Previous Next Get started

This lesson teaches you to

1. Register a Listener

You should also read

- Action Bar API Guide
- Android Design Guide

Try it out

Get the sample

ImmersiveMode sample

This lesson describes how to register a listener so that your app can get notified of system UI visibility changes. This is useful if you want to synchronize other parts of your UI with the hiding/showing of system bars.

Register a Listener

To get notified of system UI visibility changes, register an `View.OnSystemUiVisibilityChangeListener` to your view. This is typically the view you are using to control the navigation visibility.

For example, you could add this code to your activity's `onCreate()`

method:

```
View decorView = getWindow().getDecorView();
decorView.setOnSystemUiVisibilityChangeListener
        (new View.OnSystemUiVisibilityChangeListener() {
    @Override
    public void onSystemUiVisibilityChange(int visibility) {
        // Note that system bars will only be "visible" if none of
the
        // LOW_PROFILE, HIDE_NAVIGATION, or FULLSCREEN flags are
set.
        if ((visibility & View.SYSTEM_UI_FLAG_FULLSCREEN) == 0) {
            // TODO: The system bars are visible. Make any desired
            // adjustments to your UI, such as showing the action
bar or
            // other navigational controls.
        } else {
            // TODO: The system bars are NOT visible. Make any
desired
            // adjustments to your UI, such as hiding the action bar
or
            // other navigational controls.
        }
    }
});
```

It's generally good practice to keep your UI in sync with changes in system bar visibility. For example, you could use this listener to hide and show the action bar in concert with the status bar hiding and showing.

Next Get started Next class

Material Design for Developers

Previous Next Get started

Dependencies and Prerequisites

- Android 5.0 (API level 21)

Material design is a comprehensive guide for visual, motion, and interaction design across platforms and devices. To use material design in your Android apps, follow the guidelines described in the material design specification and use the new components and functionality available in Android 5.0 (API level 21).

This class shows you how to create material design apps with the following elements:

- The material theme
- Widgets for cards and lists
- Custom shadows and view clipping
- Vector drawables
- Custom animations

This class also teaches you how to maintain compatibility with versions of Android earlier than 5.0 (API level 21) when you use material design features in your app.

Lessons

Getting Started
: Learn how to update your app with material design features.

Using the Material Theme
: Learn how to apply material design styles to your app.

Creating Lists and Cards

Learn how to create lists and cards with a consistent look and feel using system widgets.

Defining Shadows and Clipping Views

Learn how to set elevation for your views to create custom shadows and how to clip views.

Working with Drawables

Learn how to create vector drawables and how to tint drawable resources.

Defining Custom Animations

Learn how to create custom animations for views and activity transitions with shared elements.

Maintaining Compatibility

Learn how to maintain compatibility with platform versions earlier than Android 5.0.

Next Get started Next class

Getting Started

Previous Next Get started

This lesson teaches you to

1. Apply the Material Theme
2. Design Your Layouts
3. Specify Elevation in Your Views
4. Create Lists and Cards
5. Customize Your Animations

You should also read

- Material design specification
- Material design on Android

To create apps with material design:

1. Review the material design specification.
2. Apply the material **theme** to your app.
3. Create your **layouts** following material design guidelines.
4. Specify the **elevation** of your views to cast shadows.
5. Use system **widgets** for lists and cards.
6. Customize the **animations** in your app.

Maintain backward compatibility

You can add many material design features to your app while maintaining compatibility with versions of Android earlier than 5.0. For more information, see Maintaining Compatibility.

Update your app with material design

To update an existing app to incorporate material design, update your layouts following material design guidelines. Also make sure to incorporate depth, touch feedback, and animations.

Create new apps with material design

If you are creating a new app with material design features, the material design guidelines provide you with a cohesive design framework. Follow those guidelines and use the new functionality in the Android framework to design and develop your app.

Apply the Material Theme

To apply the material theme in your app, specify a style that inherits from `android:Theme.Material`:

```
<!-- res/values/styles.xml -->
<resources>
  <!-- your theme inherits from the material theme -->
  <style name="AppTheme" parent="android:Theme.Material">
    <!-- theme customizations -->
  </style>
</resources>
```

The material theme provides updated system widgets that let you set their color palette and default animations for touch feedback and activity transitions. For more details, see Using the Material Theme.

Design Your Layouts

In addition to applying and customizing the material theme, your layouts should conform to the material design guidelines. When you design your layouts, pay special attention to the following:

- Baseline grids
- Keylines
- Spacing

- Touch target size
- Layout structure

Specify Elevation in Your Views

Views can cast shadows, and the elevation value of a view determines the size of its shadow and its drawing order. To set the elevation of a view, use the `android:elevation` attribute in your layouts:

```
<TextView
    android:id="@+id/my_textview"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/next"
    android:background="@color/white"
    android:elevation="5dp" />
```

The new `translationZ` property lets you create animations that reflect temporary changes in the elevation of a view. Elevation changes can be useful when responding to touch gestures.

For more details, see Defining Shadows and Clipping Views.

Create Lists and Cards

`RecyclerView` is a more pluggable version of `ListView` that supports different layout types and provides performance improvements. `CardView` lets you show pieces of information inside cards with a consistent look across apps. The following code example demonstrates how to include a `CardView` in your layout:

```
<android.support.v7.widget.CardView
    android:id="@+id/card_view"
    android:layout_width="200dp"
    android:layout_height="200dp"
    card_view:cardCornerRadius="3dp">
    ...
</android.support.v7.widget.CardView>
```

For more information, see Creating Lists and Cards.

Customize Your Animations

Android 5.0 (API level 21) includes new APIs to create custom animations in your app. For example, you can enable activity transitions and define an exit transition inside an activity:

```
public class MyActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // enable transitions

getWindow().requestFeature(Window.FEATURE_CONTENT_TRANSITIONS);
        setContentView(R.layout.activity_my);
    }

    public void onSomeButtonClicked(View view) {
        getWindow().setExitTransition(new Explode());
        Intent intent = new Intent(this, MyOtherActivity.class);
        startActivity(intent,
                      ActivityOptions

.makeSceneTransitionAnimation(this).toBundle());
    }
}
```

When you start another activity from this activity, the exit transition is activated.

To learn more about the new animation APIs, see Defining Custom Animations.

Next Get started Next class

Using the Material Theme

Previous Next Get started

This lesson teaches you to

1. Customize the Color Palette
2. Customize the Status Bar
3. Theme Individual Views

You should also read

- Material design specification
- Material design on Android

The new material theme provides:

- System widgets that let you set their color palette
- Touch feedback animations for the system widgets
- Activity transition animations

You can customize the look of the material theme according to your brand identity with a color palette you control. You can tint the action bar and the status bar using theme attributes, as shown in Figure 3.

The system widgets have a new design and touch feedback animations. You can customize the color palette, the touch feedback animations, and the activity transitions for your app.

The material theme is defined as:

- `@android:style/Theme.Material` (dark version)
- `@android:style/Theme.Material.Light` (light version)
- `@android:style/Theme.Material.Light.DarkActionBar`

For a list of material styles that you can use, see the API reference for `R.style`.

Figure 1. Dark material theme

Figure 2. Light material theme

Note: The material theme is only available in Android 5.0 (API level 21) and above. The v7 Support Libraries provide themes with material design styles for some widgets and support for customizing the color palette. For more information, see Maintaining Compatibility.

Customize the Color Palette

To customize the theme's base colors to fit your brand, define your custom colors using theme attributes when you inherit from the material theme:

```
<resources>
  <!-- inherit from the material theme -->
  <style name="AppTheme" parent="android:Theme.Material">
    <!-- Main theme colors -->
    <!--   your app branding color for the app bar -->
    <item name="android:colorPrimary">@color/primary</item>
    <!--   darker variant for the status bar and contextual app bars -->
    <item name="android:colorPrimaryDark">@color/primary_dark</item>
    <!--   theme UI controls like checkboxes and text fields -->
    <item name="android:colorAccent">@color/accent</item>
  </style>
</resources>
```

Customize the Status Bar

The material theme lets you easily customize the status bar, so you can specify a color that fits your brand and provides enough contrast to show the white status icons. To set a custom color for the status bar, use the `android:statusBarColor` attribute when you extend the material theme. By default, `android:statusBarColor` inherits the value of `android:colorPrimaryDark`.

You can also draw behind the status bar yourself. For example, if you want to show the status bar transparently over a photo, with a subtle dark gradient to ensure the white status icons are visible. To do so, set the `android:statusBarColor` attribute to `@android:color/transparent` and adjust the window flags as required. You can also use the `Window.setStatusBarColor()` method for animations or fading.

Figure 3. Customizing the material theme.

Note: The status bar should almost always have a clear delineation from the primary toolbar, except for cases where you show edge-to-edge rich imagery or media content behind these bars and when you use a gradient to ensure that the icons are still visible.

When you customize the navigation and status bars, either make them both transparent or modify only the status bar. The navigation bar should remain black in all other cases.

Theme Individual Views

Elements in XML layout definitions can specify the `android:theme` attribute, which references a theme resource. This attribute modifies the theme for the element and any child elements, which is useful for altering theme color palettes in a specific portion of an interface.

Next Get started Next class

Creating Lists and Cards

Previous Next Get started

This lesson teaches you to

1. Create Lists
2. Create Cards
3. Add Dependencies

You should also read

- Material design specification
- Material design on Android

To create complex lists and cards with material design styles in your apps, you can use the `RecyclerView` and `CardView` widgets.

Create Lists

The `RecyclerView` widget is a more advanced and flexible version of `ListView`. This widget is a container for displaying large data sets that can be scrolled very efficiently by maintaining a limited number of views. Use the `RecyclerView` widget when you have data collections whose elements change at runtime based on user action or network events.

The `RecyclerView` class simplifies the display and handling of large data sets by providing:

- Layout managers for positioning items
- Default animations for common item operations, such as removal or addition of items

You also have the flexibility to define custom layout managers and animations for `RecyclerView` widgets.

Figure 1. The `RecyclerView` widget.

To use the `RecyclerView` widget, you have to specify an adapter and a layout manager. To create an adapter, extend the `RecyclerView.Adapter` class. The details of the implementation depend on the specifics of your dataset and the type of views. For more information, see the examples below.

A **layout manager** positions item views inside a `RecyclerView` and determines when to reuse item views that are no longer visible to the user. To reuse (or *recycle*) a view, a layout manager may ask the adapter to replace the contents of the view with a different element from the dataset. Recycling views in this manner improves performance by avoiding the creation of unnecessary views or performing expensive `findViewById()` lookups.

`RecyclerView` provides these built-in layout managers:

- `LinearLayoutManager` shows items in a vertical or horizontal scrolling list.
- `GridLayoutManager` shows items

Figure 2 - Lists with `RecyclerView`.

in a grid.

- `StaggeredGridLayoutManager` shows items in a staggered grid.

To create a custom layout manager, extend the `RecyclerView.LayoutManager` class.

Animations

Animations for adding and removing items are enabled by default in `RecyclerView`. To customize these animations, extend the `RecyclerView.ItemAnimator` class and use the `RecyclerView.setItemAnimator()` method.

Examples

The following code example demonstrates how to add the `RecyclerView` to a layout:

```
<!-- A RecyclerView with some commonly used attributes -->
<android.support.v7.widget.RecyclerView
    android:id="@+id/my_recycler_view"
    android:scrollbars="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

Once you have added a `RecyclerView` widget to your layout, obtain a handle to the object, connect it to a layout manager, and attach an adapter for the data to be displayed:

```
public class MyActivity extends Activity {
    private RecyclerView mRecyclerView;
    private RecyclerView.Adapter mAdapter;
    private RecyclerView.LayoutManager mLayoutManager;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.my_activity);
        mRecyclerView = (RecyclerView)
findViewById(R.id.my_recycler_view);

        // use this setting to improve performance if you know that
changes
        // in content do not change the layout size of the
RecyclerView
        mRecyclerView.setHasFixedSize(true);

        // use a linear layout manager
```

```
        mLayoutManager = new LinearLayoutManager(this);
        mRecyclerView.setLayoutManager(mLayoutManager);

        // specify an adapter (see also next example)
        mAdapter = new MyAdapter(myDataset);
        mRecyclerView.setAdapter(mAdapter);
    }
    ...
}
```

The adapter provides access to the items in your data set, creates views for items, and replaces the content of some of the views with new data items when the original item is no longer visible. The following code example shows a simple implementation for a data set that consists of an array of strings displayed using `TextView` widgets:

```
public class MyAdapter extends
RecyclerView.Adapter<MyAdapter.ViewHolder> {
    private String[] mDataset;

    // Provide a reference to the views for each data item
    // Complex data items may need more than one view per item, and
    // you provide access to all the views for a data item in a
view holder
    public static class ViewHolder extends RecyclerView.ViewHolder {
        // each data item is just a string in this case
        public TextView mTextView;
        public ViewHolder(TextView v) {
            super(v);
            mTextView = v;
        }
    }

    // Provide a suitable constructor (depends on the kind of
dataset)
    public MyAdapter(String[] myDataset) {
        mDataset = myDataset;
    }

    // Create new views (invoked by the layout manager)
    @Override
    public MyAdapter.ViewHolder onCreateViewHolder(ViewGroup parent,
                                                   int viewType) {
        // create a new view
        View v = LayoutInflater.from(parent.getContext())
                                .inflate(R.layout.my_text_view,
parent, false);
        // set the view's size, margins, paddings and layout
parameters
        ...
        ViewHolder vh = new ViewHolder(v);
        return vh;
    }

    // Replace the contents of a view (invoked by the layout
manager)
    @Override
```

```
    public void onBindViewHolder(ViewHolder holder, int position) {
        // - get element from your dataset at this position
        // - replace the contents of the view with that element
        holder.mTextView.setText(mDataset[position]);

    }

    // Return the size of your dataset (invoked by the layout
manager)
    @Override
    public int getItemCount() {
        return mDataset.length;
    }
}
```

Create Cards

`CardView` extends the `FrameLayout` class and lets you show information inside cards that have a consistent look across the platform. `CardView` widgets can have shadows and rounded corners.

To create a card with a shadow, use the `card_view:cardElevation` attribute. `CardView` uses real elevation and dynamic shadows on Android 5.0 (API level 21) and above and falls back to a programmatic shadow implementation on earlier versions. For more information, see Maintaining Compatibility.

Figure 3. Card examples.

Use these properties to customize the appearance of the `CardView` widget:

- To set the corner radius in your layouts, use the `card_view:cardCornerRadius` attribute.
- To set the corner radius in your code, use the `CardView.setRadius` method.
- To set the background color of a card, use the `card_view:cardBackgroundColor` attribute.

The following code example shows you how to include a `CardView` widget

in your layout:

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:card_view="http://schemas.android.com/apk/res-auto"
    ... >
    <!-- A CardView that contains a TextView -->
    <android.support.v7.widget.CardView
        xmlns:card_view="http://schemas.android.com/apk/res-auto"
        android:id="@+id/card_view"
        android:layout_gravity="center"
        android:layout_width="200dp"
        android:layout_height="200dp"
        card_view:cardCornerRadius="4dp">

        <TextView
            android:id="@+id/info_text"
            android:layout_width="match_parent"
            android:layout_height="match_parent" />
    </android.support.v7.widget.CardView>
</LinearLayout>
```

For more information, see the API reference for `CardView`.

Add Dependencies

The `RecyclerView` and `CardView` widgets are part of the v7 Support Libraries. To use these widgets in your project, add these Gradle dependencies to your app's module:

```
dependencies {
    ...
    compile 'com.android.support:cardview-v7:21.0.+'
    compile 'com.android.support:recyclerview-v7:21.0.+'
}
```

Next Get started Next class

Defining Shadows and Clipping Views

Previous Next Get started

This lesson teaches you to

1. Assign Elevation to Your Views
2. Customize View Shadows and Outlines
3. Clip Views

You should also read

- Material design specification
- Material design on Android

Material design introduces elevation for UI elements. Elevation helps users understand the relative importance of each element and focus their attention to the task at hand.

The elevation of a view, represented by the Z property, determines the visual appearance of its shadow: views with higher Z values cast larger, softer shadows. Views with higher Z values occlude views with lower Z values; however, the Z value of a view does not affect the view's size.

Shadows are drawn by the parent of the elevated view, and thus subject to standard view clipping, clipped by the parent by default.

Elevation is also useful to create animations where widgets temporarily rise above the view plane when performing some action.

For more information about elevation in material design, see Objects in 3D space.

Assign Elevation to Your Views

The Z value for a view has two components:

- Elevation: The static component.
- Translation: The dynamic component used for animations.

```
Z = elevation + translationZ
```

Figure 1 - Shadows for different view elevations.

To set the elevation of a view in a layout definition, use the `android:elevation` attribute. To set the elevation of a view in the code of an activity, use the `View.setElevation()` method.

To set the translation of a view, use the `View.setTranslationZ()` method.

The new `ViewPropertyAnimator.z()` and `ViewPropertyAnimator.translationZ()` methods enable you to easily animate the elevation of views. For more information, see the API reference for `ViewPropertyAnimator` and the Property Animation developer guide.

You can also use a `StateListAnimator` to specify these animations in a declarative way. This is especially useful for cases where state changes

trigger animations, like when a user presses a button. For more information, see Animate View State Changes.

The Z values are measured in dp (density-independent pixels).

Customize View Shadows and Outlines

The bounds of a view's background drawable determine the default shape of its shadow. **Outlines** represent the outer shape of a graphics object and define the ripple area for touch feedback.

Consider this view, defined with a background drawable:

```
<TextView
    android:id="@+id/myview"
    ...
    android:elevation="2dp"
    android:background="@drawable/myrect" />
```

The background drawable is defined as a rectangle with rounded corners:

```
<!-- res/drawable/myrect.xml -->
<shape xmlns:android="http://schemas.android.com/apk/res/android"
       android:shape="rectangle">
    <solid android:color="#42000000" />
    <corners android:radius="5dp" />
</shape>
```

The view casts a shadow with rounded corners, since the background drawable defines the view's outline. Providing a custom outline overrides the default shape of a view's shadow.

To define a custom outline for a view in your code:

1. Extend the `ViewOutlineProvider` class.
2. Override the `getOutline()` method.
3. Assign the new outline provider to your view with the `View.setOutlineProvider()` method.

You can create oval and rectangular outlines with rounded corners using the methods in the `Outline` class. The default outline provider for views obtains the outline from the view's background. To prevent a view from casting a shadow, set its outline provider to `null`.

Clip Views

Clipping views enables you to easily change the shape of a view. You can clip views for consistency with other design elements or to change the shape of a view in response to user input. You can clip a view to its outline area using the `View.setClipToOutline()` method or the `android:clipToOutline` attribute. Only rectangle, circle, and round rectangle outlines support clipping, as determined by the `Outline.canClip()` method.

To clip a view to the shape of a drawable, set the drawable as the background of the view (as shown above) and call the `View.setClipToOutline()` method.

Clipping views is an expensive operation, so don't animate the shape you use to clip a view. To achieve this effect, use the Reveal Effect animation.

Next Get started Next class

Working with Drawables

Previous Next Get started

This lesson teaches you to

1. Tint Drawable Resources
2. Extract Prominent Colors from an Image
3. Create Vector Drawables

You should also read

- Material design specification
- Material design on Android

The following capabilities for drawables help you implement material design in your apps:

- Drawable tinting
- Prominent color extraction
- Vector drawables

This lesson shows you how to use these features in your app.

Tint Drawable Resources

With Android 5.0 (API level 21) and above, you can tint bitmaps and nine-patches defined as alpha masks. You can tint them with color resources or theme attributes that resolve to color resources (for example, `?android:attr/colorPrimary`). Usually, you create these assets only once and color them automatically to match your theme.

You can apply a tint to `BitmapDrawable`, `NinePatchDrawable` or

`VectorDrawable` objects with the `setTint()` method. You can also set the tint color and mode in your layouts with the `android:tint` and `android:tintMode` attributes.

Extract Prominent Colors from an Image

The Android Support Library r21 and above includes the `Palette` class, which lets you extract prominent colors from an image. This class extracts the following prominent colors:

- Vibrant
- Vibrant dark
- Vibrant light
- Muted
- Muted dark
- Muted light

To extract these colors, pass a `Bitmap` object to the `Palette.generate()` static method in the background thread where you load your images. If you can't use that thread, call the `Palette.generateAsync()` method and provide a listener instead.

You can retrieve the prominent colors from the image using the getter methods in the `Palette` class, such as `Palette.getVibrantColor`.

To use the `Palette` class in your project, add the following Gradle dependency to your app's module:

```
dependencies {
    ...
    compile 'com.android.support:palette-v7:21.0.0'
}
```

For more information, see the API reference for the `Palette` class.

Create Vector Drawables

Video

Android Vector Graphics

In Android 5.0 (API Level 21) and above, you can define vector drawables, which scale without losing definition. You need only one asset file for a vector image, as opposed to an asset file for each screen density in the case of bitmap images. To create a vector image, you define the details of the shape inside a `<vector>` XML element.

The following example defines a vector image with the shape of a heart:

```
<!-- res/drawable/heart.xml -->
<vector xmlns:android="http://schemas.android.com/apk/res/android"
    <!-- intrinsic size of the drawable -->
    android:height="256dp"
    android:width="256dp"
    <!-- size of the virtual canvas -->
    android:viewportWidth="32"
    android:viewportHeight="32">

  <!-- draw a path -->
  <path android:fillColor="#8fff"
      android:pathData="M20.5,9.5
                        c-1.955,0,-3.83,1.268,-4.5,3
                        c-0.67,-1.732,-2.547,-3,-4.5,-3
                        C8.957,9.5,7,11.432,7,14
                        c0,3.53,3.793,6.257,9,11.5
                        c5.207,-5.242,9,-7.97,9,-11.5
                        C25,11.432,23.043,9.5,20.5,9.5z" />
</vector>
```

Vector images are represented in Android as `VectorDrawable` objects. For more information about the `pathData` syntax, see the SVG Path reference. For more information about animating the properties of vector drawables, see Animating Vector Drawables.

Next Get started Next class

Defining Custom Animations

Previous Next Get started

This lesson teaches you to

1. Customize Touch Feedback
2. Use the Reveal Effect
3. Customize Activity Transitions
4. Animate View State Changes
5. Animate Vector Drawables

You should also read

- Material design specification
- Material design on Android

Animations in material design give users feedback on their actions and provide visual continuity as users interact with your app. The material theme provides some default animations for buttons and activity transitions, and Android 5.0 (API level 21) and above lets you customize these animations and create new ones:

- Touch feedback
- Circular Reveal
- Activity transitions
- Curved motion
- View state changes

Customize Touch Feedback

Touch feedback in material design provides an instantaneous visual confirmation at the point of contact when users interact with UI

elements. The default touch feedback animations for buttons use the new `RippleDrawable` class, which transitions between different states with a ripple effect.

In most cases, you should apply this functionality in your view XML by specifying the view background as:

- `?android:attr/selectableItemBackground` for a bounded ripple.
- `?android:attr/selectableItemBackgroundBorderless` for a ripple that extends beyond the view. It will be drawn upon, and bounded by, the nearest parent of the view with a non-null background.

Note: `selectableItemBackgroundBorderless` is a new attribute introduced in API level 21.

Alternatively, you can define a `RippleDrawable` as an XML resource using the `ripple` element.

You can assign a color to `RippleDrawable` objects. To change the default touch feedback color, use the theme's `android:colorControlHighlight` attribute.

For more information, see the API reference for the `RippleDrawable` class.

Use the Reveal Effect

Reveal animations provide users visual continuity when you show or hide a group of UI elements. The `ViewAnimationUtils.createCircularReveal()` method enables you to animate a clipping circle to reveal or hide a view.

To reveal a previously invisible view using this effect:

```
// previously invisible view
View myView = findViewById(R.id.my_view);

// get the center for the clipping circle
int cx = myView.getWidth() / 2;
int cy = myView.getHeight() / 2;

// get the final radius for the clipping circle
int finalRadius = Math.max(myView.getWidth(), myView.getHeight());
```

```
// create the animator for this view (the start radius is zero)
Animator anim =
    ViewAnimationUtils.createCircularReveal(myView, cx, cy, 0,
finalRadius);

// make the view visible and start the animation
myView.setVisibility(View.VISIBLE);
anim.start();
```

To hide a previously visible view using this effect:

```
// previously visible view
final View myView = findViewById(R.id.my_view);

// get the center for the clipping circle
int cx = myView.getWidth() / 2;
int cy = myView.getHeight() / 2;

// get the initial radius for the clipping circle
int initialRadius = myView.getWidth();

// create the animation (the final radius is zero)
Animator anim =
    ViewAnimationUtils.createCircularReveal(myView, cx, cy,
initialRadius, 0);

// make the view invisible when the animation is done
anim.addListener(new AnimatorListenerAdapter() {
    @Override
    public void onAnimationEnd(Animator animation) {
        super.onAnimationEnd(animation);
        myView.setVisibility(View.INVISIBLE);
    }
});

// start the animation
anim.start();
```

Customize Activity Transitions

Figure 1 - A transition with shared elements.

To replay the movie, click on the device screen

Activity transitions in material design apps provide visual connections between different states through motion and transformations between common elements. You can specify custom animations for enter and exit transitions and for transitions of shared elements between activities.

- An **enter** transition determines how views in an activity enter the scene. For example, in the *explode* enter transition, the views enter

the scene from the outside and fly in towards the center of the screen.
- An **exit** transition determines how views in an activity exit the scene. For example, in the *explode* exit transition, the views exit the scene away from the center.
- A **shared elements** transition determines how views that are shared between two activities transition between these activities. For example, if two activities have the same image in different positions and sizes, the *changeImageTransform* shared element transition translates and scales the image smoothly between these activities.

Android 5.0 (API level 21) supports these enter and exit transitions:

- *explode* - Moves views in or out from the center of the scene.
- *slide* - Moves views in or out from one of the edges of the scene.
- *fade* - Adds or removes a view from the scene by changing its opacity.

Any transition that extends the `Visibility` class is supported as an enter or exit transition. For more information, see the API reference for the `Transition` class.

Android 5.0 (API level 21) also supports these shared elements transitions:

- *changeBounds* - Animates the changes in layout bounds of target views.
- *changeClipBounds* - Animates the changes in clip bounds of target views.
- *changeTransform* - Animates the changes in scale and rotation of target views.
- *changeImageTransform* - Animates changes in size and scale of target images.

When you enable activity transitions in your app, the default cross-fading transition is activated between the entering and exiting activities.

Figure 2 - A scene transition with one shared element.

Specify custom transitions

First, enable window content transitions with the `android:windowContentTransitions` attribute when you define a style that inherits from the material theme. You can also specify enter, exit, and shared element transitions in your style definition:

```
<style name="BaseAppTheme" parent="android:Theme.Material">
  <!-- enable window content transitions -->
  <item name="android:windowContentTransitions">true</item>

  <!-- specify enter and exit transitions -->
  <item
name="android:windowEnterTransition">@transition/explode</item>
  <item
name="android:windowExitTransition">@transition/explode</item>

  <!-- specify shared element transitions -->
  <item name="android:windowSharedElementEnterTransition">
    @transition/change_image_transform</item>
  <item name="android:windowSharedElementExitTransition">
    @transition/change_image_transform</item>
</style>
```

The `change_image_transform` transition in this example is defined as follows:

```
<!-- res/transition/change_image_transform.xml -->
<!-- (see also Shared Transitions below) -->
<transitionSet
xmlns:android="http://schemas.android.com/apk/res/android">
  <changeImageTransform/>
</transitionSet>
```

The `changeImageTransform` element corresponds to the `ChangeImageTransform` class. For more information, see the API reference for `Transition`.

To enable window content transitions in your code instead, call the `Window.requestFeature()` method:

```
// inside your activity (if you did not enable transitions in your
theme)
getWindow().requestFeature(Window.FEATURE_CONTENT_TRANSITIONS);

// set an exit transition
getWindow().setExitTransition(new Explode());
```

To specify transitions in your code, call these methods with a `Transition` object:

- `Window.setEnterTransition()`
- `Window.setExitTransition()`
- `Window.setSharedElementEnterTransition()`
- `Window.setSharedElementExitTransition()`

The `setExitTransition()` and `setSharedElementExitTransition()` methods define the exit transition for the calling activity. The `setEnterTransition()` and `setSharedElementEnterTransition()` methods define the enter transition for the called activity.

To get the full effect of a transition, you must enable window content transitions on both the calling and called activities. Otherwise, the calling activity will start the exit transition, but then you'll see a window transition (like scale or fade).

To start an enter transition as soon as possible, use the

`Window.setAllowEnterTransitionOverlap()` method on the called activity. This lets you have more dramatic enter transitions.

Start an activity using transitions

If you enable transitions and set an exit transition for an activity, the transition is activated when you launch another activity as follows:

```
startActivity(intent,

ActivityOptions.makeSceneTransitionAnimation(this).toBundle());
```

If you have set an enter transition for the second activity, the transition is also activated when the activity starts. To disable transitions when you start another activity, provide a `null` options bundle.

Start an activity with a shared element

To make a screen transition animation between two activities that have a shared element:

1. Enable window content transitions in your theme.
2. Specify a shared elements transition in your style.
3. Define your transition as an XML resource.
4. Assign a common name to the shared elements in both layouts with the `android:transitionName` attribute.
5. Use the `ActivityOptions.makeSceneTransitionAnimation()` method.

```
// get the element that receives the click event
final View imgContainerView = findViewById(R.id.img_container);

// get the common element for the transition in this activity
final View androidRobotView = findViewById(R.id.image_small);

// define a click listener
imgContainerView.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Intent intent = new Intent(this, Activity2.class);
        // create the transition animation - the images in the
layouts
        // of both activities are defined with
android:transitionName="robot"
        ActivityOptions options = ActivityOptions
            .makeSceneTransitionAnimation(this, androidRobotView,
"robot");
```

```
        // start the new activity
        startActivity(intent, options.toBundle());
    }
});
```

For shared dynamic views that you generate in your code, use the `View.setTransitionName()` method to specify a common element name in both activities.

To reverse the scene transition animation when you finish the second activity, call the `Activity.finishAfterTransition()` method instead of `Activity.finish()`.

Start an activity with multiple shared elements

To make a scene transition animation between two activities that have more than one shared element, define the shared elements in both layouts with the `android:transitionName` attribute (or use the `View.setTransitionName()` method in both activities), and create an `ActivityOptions` object as follows:

```
ActivityOptions options =
ActivityOptions.makeSceneTransitionAnimation(this,
        Pair.create(view1, "agreedName1"),
        Pair.create(view2, "agreedName2"));
```

Use Curved Motion

Animations in material design rely on curves for time interpolation and spatial movement patterns. With Android 5.0 (API level 21) and above, you can define custom timing curves and curved motion patterns for animations.

The `PathInterpolator` class is a new interpolator based on a Bézier curve or a `Path` object. This interpolator specifies a motion curve in a 1x1 square, with anchor points at (0,0) and (1,1) and control points as specified using the constructor arguments. You can also define a path interpolator as an XML resource:

```
<pathInterpolator
xmlns:android="http://schemas.android.com/apk/res/android"
    android:controlX1="0.4"
```

```
    android:controlY1="0"
    android:controlX2="1"
    android:controlY2="1"/>
```

The system provides XML resources for the three basic curves in the material design specification:

- `@interpolator/fast_out_linear_in.xml`
- `@interpolator/fast_out_slow_in.xml`
- `@interpolator/linear_out_slow_in.xml`

You can pass a `PathInterpolator` object to the `Animator.setInterpolator()` method.

The `ObjectAnimator` class has new constructors that enable you to animate coordinates along a path using two or more properties at once. For example, the following animator uses a `Path` object to animate the X and Y properties of a view:

```
ObjectAnimator mAnimator;
mAnimator = ObjectAnimator.ofFloat(view, View.X, View.Y, path);
...
mAnimator.start();
```

Animate View State Changes

The `StateListAnimator` class lets you define animators that run when the state of a view changes. The following example shows how to define an `StateListAnimator` as an XML resource:

```
<!-- animate the translationZ property of a view when pressed -->
<selector xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:state_pressed="true">
    <set>
      <objectAnimator android:propertyName="translationZ"
        android:duration="@android:integer/config_shortAnimTime"
        android:valueTo="2dp"
        android:valueType="floatType"/>
        <!-- you could have other objectAnimator elements
             here for "x" and "y", or other properties -->
    </set>
  </item>
  <item android:state_enabled="true"
    android:state_pressed="false"
    android:state_focused="true">
    <set>
      <objectAnimator android:propertyName="translationZ"
```

```
        android:duration="100"
        android:valueTo="0"
        android:valueType="floatType"/>
    </set>
  </item>
</selector>
```

To attach custom view state animations to a view, define an animator using the `selector` element in an XML resource file as in this example, and assign it to your view with the `android:stateListAnimator` attribute. To assign a state list animator to a view in your code, use the `AnimationInflater.loadStateListAnimator()` method, and assign the animator to your view with the `View.setStateListAnimator()` method.

When your theme extends the material theme, buttons have a Z animation by default. To avoid this behavior in your buttons, set the `android:stateListAnimator` attribute to `@null`.

The `AnimatedStateListDrawable` class lets you create drawables that show animations between state changes of the associated view. Some of the system widgets in Android 5.0 use these animations by default. The following example shows how to define an `AnimatedStateListDrawable` as an XML resource:

```
<!-- res/drawable/myanimstatedrawable.xml -->
<animated-selector
    xmlns:android="http://schemas.android.com/apk/res/android">

    <!-- provide a different drawable for each state-->
    <item android:id="@+id/pressed"
android:drawable="@drawable/drawableP"
        android:state_pressed="true"/>
    <item android:id="@+id/focused"
android:drawable="@drawable/drawableF"
        android:state_focused="true"/>
    <item android:id="@id/default"
        android:drawable="@drawable/drawableD"/>

    <!-- specify a transition -->
    <transition android:fromId="@+id/default"
android:toId="@+id/pressed">
        <animation-list>
            <item android:duration="15"
android:drawable="@drawable/dt1"/>
            <item android:duration="15"
android:drawable="@drawable/dt2"/>
            ...
        </animation-list>
    </transition>
    ...
</animated-selector>
```

Animate Vector Drawables

Vector Drawables are scalable without losing definition. The `AnimatedVectorDrawable` class lets you animate the properties of a vector drawable.

You normally define animated vector drawables in three XML files:

- A vector drawable with the `<vector>` element in `res/drawable/`
- An animated vector drawable with the `<animated-vector>` element in `res/drawable/`
- One or more object animators with the `<objectAnimator>` element in `res/anim/`

Animated vector drawables can animate the attributes of the `<group>` and `<path>` elements. The `<group>` elements defines a set of paths or subgroups, and the `<path>` element defines paths to be drawn.

When you define a vector drawable that you want to animate, use the `android:name` attribute to assign a unique name to groups and paths, so you can refer to them from your animator definitions. For example:

```
<!-- res/drawable/vectordrawable.xml -->
<vector xmlns:android="http://schemas.android.com/apk/res/android"
    android:height="64dp"
    android:width="64dp"
    android:viewportHeight="600"
    android:viewportWidth="600">
    <group
        android:name="rotationGroup"
        android:pivotX="300.0"
        android:pivotY="300.0"
        android:rotation="45.0" >
        <path
            android:name="v"
            android:fillColor="#000000"
            android:pathData="M300,70 l 0,-70 70,70 0,0 -70,70z" />
    </group>
</vector>
```

The animated vector drawable definition refers to the groups and paths in the vector drawable by their names:

```
<!-- res/drawable/animvectordrawable.xml -->
```

```
<animated-vector
xmlns:android="http://schemas.android.com/apk/res/android"
  android:drawable="@drawable/vectordrawable" >
    <target
        android:name="rotationGroup"
        android:animation="@anim/rotation" />
    <target
        android:name="v"
        android:animation="@anim/path_morph" />
</animated-vector>
```

The animation definitions represent `ObjectAnimator` or `AnimatorSet` objects. The first animator in this example rotates the target group 360 degrees:

```
<!-- res/anim/rotation.xml -->
<objectAnimator
    android:duration="6000"
    android:propertyName="rotation"
    android:valueFrom="0"
    android:valueTo="360" />
```

The second animator in this example morphs the vector drawable's path from one shape to another. Both paths must be compatible for morphing: they must have the same number of commands and the same number of parameters for each command.

```
<!-- res/anim/path_morph.xml -->
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <objectAnimator
        android:duration="3000"
        android:propertyName="pathData"
        android:valueFrom="M300,70 l 0,-70 70,70 0,0   -70,70z"
        android:valueTo="M300,70 l 0,-70 70,0  0,140 -70,0 z"
        android:valueType="pathType" />
</set>
```

For more information, see the API reference for `AnimatedVectorDrawable`.

Next Get started Next class

Maintaining Compatibility

Previous Next Get started

This lesson teaches you to

1. Define Alternative Styles
2. Provide Alternative Layouts
3. Use the Support Library
4. Check the System Version

You should also read

- Material design specification
- Material design on Android

Some material design features like the material theme and custom activity transitions are only available on Android 5.0 (API level 21) and above. However, you can design your apps to make use of these features when running on devices that support material design and still be compatible with devices running previous releases of Android.

Define Alternative Styles

You can configure your app to use the material theme on devices that support it and revert to an older theme on devices running earlier versions of Android:

1. Define a theme that inherits from an older theme (like Holo) in `res/values/styles.xml`.
2. Define a theme with the same name that inherits from the material theme in `res/values-v21/styles.xml`.
3. Set this theme as your app's theme in the manifest file.

Note: If your app uses the material theme but does not provide an alternative theme in this manner, your app will not run on versions of Android earlier than 5.0.

Provide Alternative Layouts

If the layouts that you design according to the material design guidelines do not use any of the new XML attributes introduced in Android 5.0 (API level 21), they will work on previous versions of Android. Otherwise, you can provide alternative layouts. You can also provide alternative layouts to customize how your app looks on earlier versions of Android.

Create your layout files for Android 5.0 (API level 21) inside `res/layout-v21/` and your alternative layout files for earlier versions of Android inside `res/layout/`. For example, `res/layout/my_activity.xml` is an alternative layout for `res/layout-v21/my_activity.xml`.

To avoid duplication of code, define your styles inside `res/values/`, modify the styles in `res/values-v21/` for the new APIs, and use style inheritance, defining base styles in `res/values/` and inheriting from those in `res/values-v21/`.

Use the Support Library

The v7 Support Libraries r21 and above includes the following material design features:

- Material design styles for some system widgets when you apply one of the `Theme.AppCompat` themes.
- Color palette theme attributes in the `Theme.AppCompat` themes.
- The `RecyclerView` widget to display data collections.
- The `CardView` widget to create cards.
- The `Palette` class to extract prominent colors from images.

System widgets

The `Theme.AppCompat` themes provide material design styles for these widgets:

- `EditText`
- `Spinner`
- `CheckBox`
- `RadioButton`
- `SwitchCompat`
- `CheckedTextView`

Color Palette

To obtain material design styles and customize the color palette with the Android v7 Support Library, apply one of the `Theme.AppCompat` themes:

```
<!-- extend one of the Theme.AppCompat themes -->
<style name="Theme.MyTheme" parent="Theme.AppCompat.Light">
    <!-- customize the color palette -->
    <item name="colorPrimary">@color/material_blue_500</item>
    <item name="colorPrimaryDark">@color/material_blue_700</item>
    <item name="colorAccent">@color/material_green_A200</item>
</style>
```

Lists and Cards

The `RecyclerView` and `CardView` widgets are available in earlier versions of Android through the Android v7 Support Library with these limitations:

- `CardView` falls back to a programmatic shadow implementation using additional padding.
- `CardView` does not clip its children views that intersect with rounded corners.

Dependencies

To use these features in versions of Android earlier than 5.0 (API level 21), include the Android v7 Support Library in your project as a Gradle dependency:

```
dependencies {
    compile 'com.android.support:appcompat-v7:21.0.+'
    compile 'com.android.support:cardview-v7:21.0.+'
```

```
    compile 'com.android.support:recyclerview-v7:21.0.+'
}
```

Check the System Version

The following features are available only in Android 5.0 (API level 21) and above:

- Activity transitions
- Touch feedback
- Reveal animations
- Path-based animations
- Vector drawables
- Drawable tinting

To preserve compatibility with earlier versions of Android, check the system `version` at runtime before you invoke the APIs for any of these features:

```
// Check if we're running on Android 5.0 or higher
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
    // Call some material design APIs here
} else {
    // Implement this feature without material design
}
```

Note: To specify which versions of Android your app supports, use the `android:minSdkVersion` and `android:targetSdkVersion` attributes in your manifest file. To use the material design features in Android 5.0, set the `android:targetSdkVersion` attribute to `21`. For more information, see the <uses-sdk> API guide.

Next Get started Next class

Best Practices for User Input

These classes cover various subjects of user input, such as touch screen gestures and text input through on-screen input methods and hardware keyboards.

Using Touch Gestures

Previous Next Get started

Dependencies and prerequisites

- Android 1.6 (API Level 4) or higher

You should also read

- Input Events API Guide
- Sensors Overview
- Making the View Interactive
- Design Guide for Gestures
- Design Guide for Touch Feedback

Try it out

Download the sample

InteractiveChart.zip

This class describes how to write apps that allow users to interact with an app via touch gestures. Android provides a variety of APIs to help you create and detect gestures.

Although your app should not depend on touch gestures for basic behaviors (since the gestures may not be available to all users in all contexts), adding touch-based interaction to your app can greatly increase its usefulness and appeal.

To provide users with a consistent, intuitive experience, your app should follow the accepted Android conventions for touch gestures. The Gestures

design guide shows you how to use common gestures in Android apps. Also see the Design Guide for Touch Feedback.

Lessons

Detecting Common Gestures
: Learn how to detect basic touch gestures such as scrolling, flinging, and double-tapping, using `GestureDetector`.

Tracking Movement
: Learn how to track movement.

Animating a Scroll Gesture
: Learn how to use scrollers (`Scroller` or `OverScroller`) to produce a scrolling animation in response to a touch event.

Handling Multi-Touch Gestures
: Learn how to detect multi-pointer (finger) gestures.

Dragging and Scaling
: Learn how to implement touch-based dragging and scaling.

Managing Touch Events in a ViewGroup
: Learn how to manage touch events in a `ViewGroup` to ensure that touch events are correctly dispatched to their target views.

Next Get started Next class

Detecting Common Gestures

Previous Next Get started

This lesson teaches you to

1. Gather Data
2. Detect Gestures

You should also read

- Input Events API Guide
- Sensors Overview
- Making the View Interactive
- Design Guide for Gestures
- Design Guide for Touch Feedback

Try it out

Download the sample

InteractiveChart.zip

A "touch gesture" occurs when a user places one or more fingers on the touch screen, and your application interprets that pattern of touches as a particular gesture. There are correspondingly two phases to gesture detection:

1. Gathering data about touch events.
2. Interpreting the data to see if it meets the criteria for any of the gestures your app supports.

Support Library Classes

The examples in this lesson use the `GestureDetectorCompat` and `MotionEventCompat` classes. These classes are in the Support Library. You should use Support Library classes where possible to provide compatibility with devices running Android 1.6 and higher. Note that `MotionEventCompat` is *not* a replacement for the `MotionEvent` class. Rather, it provides static utility methods to which you pass your `MotionEvent` object in order to receive the desired action associated with that event.

Gather Data

When a user places one or more fingers on the screen, this triggers the callback `onTouchEvent()` on the View that received the touch events. For each sequence of touch events (position, pressure, size, addition of another finger, etc.) that is ultimately identified as a gesture, `onTouchEvent()` is fired several times.

The gesture starts when the user first touches the screen, continues as the system tracks the position of the user's finger(s), and ends by capturing the final event of the user's fingers leaving the screen. Throughout this interaction, the `MotionEvent` delivered to `onTouchEvent()` provides the details of every interaction. Your app can use the data provided by the `MotionEvent` to determine if a gesture it cares about happened.

Capturing touch events for an Activity or View

To intercept touch events in an Activity or View, override the `onTouchEvent()` callback.

The following snippet uses `getActionMasked()` to extract the action the user performed from the `event` parameter. This gives you the raw data you need to determine if a gesture you care about occurred:

```
public class MainActivity extends Activity {
...
// This example shows an Activity, but you would use the same
approach if
// you were subclassing a View.
@Override
```

```
public boolean onTouchEvent(MotionEvent event){

    int action = MotionEventCompat.getActionMasked(event);

    switch(action) {
        case (MotionEvent.ACTION_DOWN) :
            Log.d(DEBUG_TAG,"Action was DOWN");
            return true;
        case (MotionEvent.ACTION_MOVE) :
            Log.d(DEBUG_TAG,"Action was MOVE");
            return true;
        case (MotionEvent.ACTION_UP) :
            Log.d(DEBUG_TAG,"Action was UP");
            return true;
        case (MotionEvent.ACTION_CANCEL) :
            Log.d(DEBUG_TAG,"Action was CANCEL");
            return true;
        case (MotionEvent.ACTION_OUTSIDE) :
            Log.d(DEBUG_TAG,"Movement occurred outside bounds " +
                    "of current screen element");
            return true;
        default :
            return super.onTouchEvent(event);
    }
}
```

You can then do your own processing on these events to determine if a gesture occurred. This is the kind of processing you would have to do for a custom gesture. However, if your app uses common gestures such as double tap, long press, fling, and so on, you can take advantage of the `GestureDetector` class. `GestureDetector` makes it easy for you to detect common gestures without processing the individual touch events yourself. This is discussed below in Detect Gestures.

Capturing touch events for a single view

As an alternative to `onTouchEvent()`, you can attach an `View.OnTouchListener` object to any `View` object using the `setOnTouchListener()` method. This makes it possible to to listen for touch events without subclassing an existing `View`. For example:

```
View myView = findViewById(R.id.my_view);
myView.setOnTouchListener(new OnTouchListener() {
    public boolean onTouch(View v, MotionEvent event) {
        // ... Respond to touch events
        return true;
    }
});
```

Beware of creating a listener that returns `false` for the `ACTION_DOWN`

event. If you do this, the listener will not be called for the subsequent `ACTION_MOVE` and `ACTION_UP` string of events. This is because `ACTION_DOWN` is the starting point for all touch events.

If you are creating a custom View, you can override `onTouchEvent()`, as described above.

Detect Gestures

Android provides the `GestureDetector` class for detecting common gestures. Some of the gestures it supports include `onDown()`, `onLongPress()`, `onFling()`, and so on. You can use `GestureDetector` in conjunction with the `onTouchEvent()` method described above.

Detecting All Supported Gestures

When you instantiate a `GestureDetectorCompat` object, one of the parameters it takes is a class that implements the `GestureDetector.OnGestureListener` interface. `GestureDetector.OnGestureListener` notifies users when a particular touch event has occurred. To make it possible for your `GestureDetector` object to receive events, you override the View or Activity's `onTouchEvent()` method, and pass along all observed events to the detector instance.

In the following snippet, a return value of `true` from the individual `on<TouchEvent>` methods indicates that you have handled the touch event. A return value of `false` passes events down through the view stack until the touch has been successfully handled.

Run the following snippet to get a feel for how actions are triggered when you interact with the touch screen, and what the contents of the `MotionEvent` are for each touch event. You will realize how much data is being generated for even simple interactions.

```
public class MainActivity extends Activity implements
        GestureDetector.OnGestureListener,
        GestureDetector.OnDoubleTapListener{

    private static final String DEBUG_TAG = "Gestures";
    private GestureDetectorCompat mDetector;
```

```
    // Called when the activity is first created.
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // Instantiate the gesture detector with the
        // application context and an implementation of
        // GestureDetector.OnGestureListener
        mDetector = new GestureDetectorCompat(this,this);
        // Set the gesture detector as the double tap
        // listener.
        mDetector.setOnDoubleTapListener(this);
    }

    @Override
    public boolean onTouchEvent(MotionEvent event){
        this.mDetector.onTouchEvent(event);
        // Be sure to call the superclass implementation
        return super.onTouchEvent(event);
    }

    @Override
    public boolean onDown(MotionEvent event) {
        Log.d(DEBUG_TAG,"onDown: " + event.toString());
        return true;
    }

    @Override
    public boolean onFling(MotionEvent event1, MotionEvent event2,
            float velocityX, float velocityY) {
        Log.d(DEBUG_TAG, "onFling: " +
event1.toString()+event2.toString());
        return true;
    }

    @Override
    public void onLongPress(MotionEvent event) {
        Log.d(DEBUG_TAG, "onLongPress: " + event.toString());
    }

    @Override
    public boolean onScroll(MotionEvent e1, MotionEvent e2, float
distanceX,
            float distanceY) {
        Log.d(DEBUG_TAG, "onScroll: " + e1.toString()+e2.toString());
        return true;
    }

    @Override
    public void onShowPress(MotionEvent event) {
        Log.d(DEBUG_TAG, "onShowPress: " + event.toString());
    }

    @Override
    public boolean onSingleTapUp(MotionEvent event) {
        Log.d(DEBUG_TAG, "onSingleTapUp: " + event.toString());
        return true;
    }

    @Override
```

```
    public boolean onDoubleTap(MotionEvent event) {
        Log.d(DEBUG_TAG, "onDoubleTap: " + event.toString());
        return true;
    }

    @Override
    public boolean onDoubleTapEvent(MotionEvent event) {
        Log.d(DEBUG_TAG, "onDoubleTapEvent: " + event.toString());
        return true;
    }

    @Override
    public boolean onSingleTapConfirmed(MotionEvent event) {
        Log.d(DEBUG_TAG, "onSingleTapConfirmed: " +
event.toString());
        return true;
    }
}
```

Detecting a Subset of Supported Gestures

If you only want to process a few gestures, you can extend `GestureDetector.SimpleOnGestureListener` instead of implementing the `GestureDetector.OnGestureListener` interface.

`GestureDetector.SimpleOnGestureListener` provides an implementation for all of the `on<TouchEvent>` methods by returning `false` for all of them. Thus you can override only the methods you care about. For example, the snippet below creates a class that extends `GestureDetector.SimpleOnGestureListener` and overrides `onFling()` and `onDown()`.

Whether or not you use `GestureDetector.OnGestureListener`, it's best practice to implement an `onDown()` method that returns `true`. This is because all gestures begin with an `onDown()` message. If you return `false` from `onDown()`, as `GestureDetector.SimpleOnGestureListener` does by default, the system assumes that you want to ignore the rest of the gesture, and the other methods of `GestureDetector.OnGestureListener` never get called. This has the potential to cause unexpected problems in your app. The only time you should return `false` from `onDown()` is if you truly want to ignore an entire gesture.

```
public class MainActivity extends Activity {

    private GestureDetectorCompat mDetector;
```

```
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mDetector = new GestureDetectorCompat(this, new
MyGestureListener());
    }

    @Override
    public boolean onTouchEvent(MotionEvent event){
        this.mDetector.onTouchEvent(event);
        return super.onTouchEvent(event);
    }

    class MyGestureListener extends
GestureDetector.SimpleOnGestureListener {
        private static final String DEBUG_TAG = "Gestures";

        @Override
        public boolean onDown(MotionEvent event) {
            Log.d(DEBUG_TAG,"onDown: " + event.toString());
            return true;
        }

        @Override
        public boolean onFling(MotionEvent event1, MotionEvent
event2,
                float velocityX, float velocityY) {
            Log.d(DEBUG_TAG, "onFling: " +
event1.toString()+event2.toString());
            return true;
        }
    }
}
```

Next Get started Next class

Tracking Movement

Previous Next Get started

This lesson teaches you to

1. Track Velocity

You should also read

- Input Events API Guide
- Sensors Overview
- Making the View Interactive
- Design Guide for Gestures
- Design Guide for Touch Feedback

Try it out

Download the sample

InteractiveChart.zip

This lesson describes how to track movement in touch events.

A new `onTouchEvent()` is triggered with an `ACTION_MOVE` event whenever the current touch contact position, pressure, or size changes. As described in Detecting Common Gestures, all of these events are recorded in the `MotionEvent` parameter of `onTouchEvent()`.

Because finger-based touch isn't always the most precise form of interaction, detecting touch events is often based more on movement than on simple contact. To help apps distinguish between movement-based gestures (such as a swipe) and non-movement gestures (such as a

single tap), Android includes the notion of "touch slop." Touch slop refers to the distance in pixels a user's touch can wander before the gesture is interpreted as a movement-based gesture. For more discussion of this topic, see Managing Touch Events in a ViewGroup.

There are several different ways to track movement in a gesture, depending on the needs of your application. For example:

- The starting and ending position of a pointer (for example, move an on-screen object from point A to point B).
- The direction the pointer is traveling in, as determined by the x and y coordinates.
- History. You can find the size of a gesture's history by calling the `MotionEvent` method `getHistorySize()`. You can then obtain the positions, sizes, time, and pressures of each of the historical events by using the motion event's `getHistorical<Value>` methods. History is useful when rendering a trail of the user's finger, such as for touch drawing. See the `MotionEvent` reference for details.
- The velocity of the pointer as it moves across the touch screen.

Track Velocity

You could have a movement-based gesture that is simply based on the distance and/or direction the pointer traveled. But velocity often is a determining factor in tracking a gesture's characteristics or even deciding whether the gesture occurred. To make velocity calculation easier, Android provides the `VelocityTracker` class and the `VelocityTrackerCompat` class in the Support Library. `VelocityTracker` helps you track the velocity of touch events. This is useful for gestures in which velocity is part of the criteria for the gesture, such as a fling.

Here is a simple example that illustrates the purpose of the methods in the `VelocityTracker` API:

```
public class MainActivity extends Activity {
    private static final String DEBUG_TAG = "Velocity";
        ...
    private VelocityTracker mVelocityTracker = null;
    @Override
    public boolean onTouchEvent(MotionEvent event) {
```

```
        int index = event.getActionIndex();
        int action = event.getActionMasked();
        int pointerId = event.getPointerId(index);

        switch(action) {
            case MotionEvent.ACTION_DOWN:
                if(mVelocityTracker == null) {
                    // Retrieve a new VelocityTracker object to
watch the velocity of a motion.
                    mVelocityTracker = VelocityTracker.obtain();
                }
                else {
                    // Reset the velocity tracker back to its
initial state.
                    mVelocityTracker.clear();
                }
                // Add a user's movement to the tracker.
                mVelocityTracker.addMovement(event);
                break;
            case MotionEvent.ACTION_MOVE:
                mVelocityTracker.addMovement(event);
                // When you want to determine the velocity, call
                // computeCurrentVelocity(). Then call getXVelocity()
                // and getYVelocity() to retrieve the velocity for
each pointer ID.
                mVelocityTracker.computeCurrentVelocity(1000);
                // Log velocity of pixels per second
                // Best practice to use VelocityTrackerCompat where
possible.
                Log.d("", "X velocity: " +

VelocityTrackerCompat.getXVelocity(mVelocityTracker,
                        pointerId));
                Log.d("", "Y velocity: " +

VelocityTrackerCompat.getYVelocity(mVelocityTracker,
                        pointerId));
                break;
            case MotionEvent.ACTION_UP:
            case MotionEvent.ACTION_CANCEL:
                // Return a VelocityTracker object back to be re-
used by others.
                mVelocityTracker.recycle();
                break;
        }
        return true;
    }
}
```

Note: Note that you should calculate velocity after an `ACTION_MOVE` event, not after `ACTION_UP`. After an `ACTION_UP`, the X and Y velocities will be 0.

Next Get started Next class

Animating a Scroll Gesture

Previous Next Get started

This lesson teaches you to

1. Understand Scrolling Terminology
2. Implement Touch-Based Scrolling

You should also read

- Input Events API Guide
- Sensors Overview
- Making the View Interactive
- Design Guide for Gestures
- Design Guide for Touch Feedback

Try it out

Download the sample

InteractiveChart.zip

In Android, scrolling is typically achieved by using the `ScrollView` class. Any standard layout that might extend beyond the bounds of its container should be nested in a `ScrollView` to provide a scrollable view that's managed by the framework. Implementing a custom scroller should only be necessary for special scenarios. This lesson describes such a scenario: displaying a scrolling effect in response to touch gestures using *scrollers*.

You can use scrollers (`Scroller` or `OverScroller`) to collect the data you need to produce a scrolling animation in response to a touch event. They

are similar, but `OverScroller` includes methods for indicating to users that they've reached the content edges after a pan or fling gesture. The `InteractiveChart` sample uses the `EdgeEffect` class (actually the `EdgeEffectCompat` class) to display a "glow" effect when users reach the content edges.

Note: We recommend that you use `OverScroller` rather than `Scroller` for scrolling animations. `OverScroller` provides the best backward compatibility with older devices.
Also note that you generally only need to use scrollers when implementing scrolling yourself. `ScrollView` and `HorizontalScrollView` do all of this for you if you nest your layout within them.

A scroller is used to animate scrolling over time, using platform-standard scrolling physics (friction, velocity, etc.). The scroller itself doesn't actually draw anything. Scrollers track scroll offsets for you over time, but they don't automatically apply those positions to your view. It's your responsibility to get and apply new coordinates at a rate that will make the scrolling animation look smooth.

Understand Scrolling Terminology

"Scrolling" is a word that can take on different meanings in Android, depending on the context.

Scrolling is the general process of moving the viewport (that is, the 'window' of content you're looking at). When scrolling is in both the x and y axes, it's called *panning*. The sample application provided with this class, `InteractiveChart`, illustrates two different types of scrolling, dragging and flinging:

- **Dragging** is the type of scrolling that occurs when a user drags her finger across the touch screen. Simple dragging is often implemented by overriding `onScroll()` in `GestureDetector.OnGestureListener`. For more discussion of dragging, see Dragging and Scaling.
- **Flinging** is the type of scrolling that occurs when a user drags and

lifts her finger quickly. After the user lifts her finger, you generally want to keep scrolling (moving the viewport), but decelerate until the viewport stops moving. Flinging can be implemented by overriding `onFling()` in `GestureDetector.OnGestureListener`, and by using a scroller object. This is the use case that is the topic of this lesson.

It's common to use scroller objects in conjunction with a fling gesture, but they can be used in pretty much any context where you want the UI to display scrolling in response to a touch event. For example, you could override `onTouchEvent()` to process touch events directly, and produce a scrolling effect or a "snapping to page" animation in response to those touch events.

Implement Touch-Based Scrolling

This section describes how to use a scroller. The snippet shown below comes from the `InteractiveChart` sample provided with this class. It uses a `GestureDetector`, and overrides the `GestureDetector.SimpleOnGestureListener` method `onFling()`. It uses `OverScroller` to track the fling gesture. If the user reaches the content edges after the fling gesture, the app displays a "glow" effect.

Note: The `InteractiveChart` sample app displays a chart that you can zoom, pan, scroll, and so on. In the following snippet, `mContentRect` represents the rectangle coordinates within the view that the chart will be drawn into. At any given time, a subset of the total chart domain and range are drawn into this rectangular area. `mCurrentViewport` represents the portion of the chart that is currently visible in the screen. Because pixel offsets are generally treated as integers, `mContentRect` is of the type `Rect`. Because the graph domain and range are decimal/float values, `mCurrentViewport` is of the type `RectF`.

The first part of the snippet shows the implementation of `onFling()`:

```
// The current viewport. This rectangle represents the currently
visible
// chart domain and range. The viewport is the part of the app that
the
```

```
// user manipulates via touch gestures.
private RectF mCurrentViewport =
        new RectF(AXIS_X_MIN, AXIS_Y_MIN, AXIS_X_MAX, AXIS_Y_MAX);

// The current destination rectangle (in pixel coordinates) into
which the
// chart data should be drawn.
private Rect mContentRect;

private OverScroller mScroller;
private RectF mScrollerStartViewport;
...
private final GestureDetector.SimpleOnGestureListener
mGestureListener
        = new GestureDetector.SimpleOnGestureListener() {
    @Override
    public boolean onDown(MotionEvent e) {
        // Initiates the decay phase of any active edge effects.
        releaseEdgeEffects();
        mScrollerStartViewport.set(mCurrentViewport);
        // Aborts any active scroll animations and invalidates.
        mScroller.forceFinished(true);

ViewCompat.postInvalidateOnAnimation(InteractiveLineGraphView.this);
        return true;
    }
    ...
    @Override
    public boolean onFling(MotionEvent e1, MotionEvent e2,
            float velocityX, float velocityY) {
        fling((int) -velocityX, (int) -velocityY);
        return true;
    }
};

private void fling(int velocityX, int velocityY) {
    // Initiates the decay phase of any active edge effects.
    releaseEdgeEffects();
    // Flings use math in pixels (as opposed to math based on the
viewport).
    Point surfaceSize = computeScrollSurfaceSize();
    mScrollerStartViewport.set(mCurrentViewport);
    int startX = (int) (surfaceSize.x * (mScrollerStartViewport.left
-
            AXIS_X_MIN) / (
            AXIS_X_MAX - AXIS_X_MIN));
    int startY = (int) (surfaceSize.y * (AXIS_Y_MAX -
            mScrollerStartViewport.bottom) / (
            AXIS_Y_MAX - AXIS_Y_MIN));
    // Before flinging, aborts the current animation.
    mScroller.forceFinished(true);
    // Begins the animation
    mScroller.fling(
            // Current scroll position
            startX,
            startY,
            velocityX,
            velocityY,
            /*
             * Minimum and maximum scroll positions. The minimum
scroll
```

```
             * position is generally zero and the maximum scroll
position
             * is generally the content size less the screen size.
So if the
             * content width is 1000 pixels and the screen width is
200
             * pixels, the maximum scroll offset should be 800
pixels.
             */
            0, surfaceSize.x - mContentRect.width(),
            0, surfaceSize.y - mContentRect.height(),
            // The edges of the content. This comes into play when
using
            // the EdgeEffect class to draw "glow" overlays.
            mContentRect.width() / 2,
            mContentRect.height() / 2);
    // Invalidates to trigger computeScroll()
    ViewCompat.postInvalidateOnAnimation(this);
}
```

When `onFling()` calls `postInvalidateOnAnimation()`, it triggers `computeScroll()` to update the values for x and y. This is typically be done when a view child is animating a scroll using a scroller object, as in this example.

Most views pass the scroller object's x and y position directly to `scrollTo()`. The following implementation of `computeScroll()` takes a different approach—it calls `computeScrollOffset()` to get the current location of x and y. When the criteria for displaying an overscroll "glow" edge effect are met (the display is zoomed in, x or y is out of bounds, and the app isn't already showing an overscroll), the code sets up the overscroll glow effect and calls `postInvalidateOnAnimation()` to trigger an invalidate on the view:

```
// Edge effect / overscroll tracking objects.
private EdgeEffectCompat mEdgeEffectTop;
private EdgeEffectCompat mEdgeEffectBottom;
private EdgeEffectCompat mEdgeEffectLeft;
private EdgeEffectCompat mEdgeEffectRight;

private boolean mEdgeEffectTopActive;
private boolean mEdgeEffectBottomActive;
private boolean mEdgeEffectLeftActive;
private boolean mEdgeEffectRightActive;

@Override
public void computeScroll() {
    super.computeScroll();

    boolean needsInvalidate = false;

    // The scroller isn't finished, meaning a fling or programmatic
```

```
pan
    // operation is currently active.
    if (mScroller.computeScrollOffset()) {
        Point surfaceSize = computeScrollSurfaceSize();
        int currX = mScroller.getCurrX();
        int currY = mScroller.getCurrY();

        boolean canScrollX = (mCurrentViewport.left > AXIS_X_MIN
                || mCurrentViewport.right < AXIS_X_MAX);
        boolean canScrollY = (mCurrentViewport.top > AXIS_Y_MIN
                || mCurrentViewport.bottom < AXIS_Y_MAX);

        /*
         * If you are zoomed in and currX or currY is
         * outside of bounds and you're not already
         * showing overscroll, then render the overscroll
         * glow edge effect.
         */
        if (canScrollX
                && currX < 0
                && mEdgeEffectLeft.isFinished()
                && !mEdgeEffectLeftActive) {
            mEdgeEffectLeft.onAbsorb((int)
                    OverScrollerCompat.getCurrVelocity(mScroller));
            mEdgeEffectLeftActive = true;
            needsInvalidate = true;
        } else if (canScrollX
                && currX > (surfaceSize.x - mContentRect.width())
                && mEdgeEffectRight.isFinished()
                && !mEdgeEffectRightActive) {
            mEdgeEffectRight.onAbsorb((int)
                    OverScrollerCompat.getCurrVelocity(mScroller));
            mEdgeEffectRightActive = true;
            needsInvalidate = true;
        }

        if (canScrollY
                && currY < 0
                && mEdgeEffectTop.isFinished()
                && !mEdgeEffectTopActive) {
            mEdgeEffectTop.onAbsorb((int)
                    OverScrollerCompat.getCurrVelocity(mScroller));
            mEdgeEffectTopActive = true;
            needsInvalidate = true;
        } else if (canScrollY
                && currY > (surfaceSize.y - mContentRect.height())
                && mEdgeEffectBottom.isFinished()
                && !mEdgeEffectBottomActive) {
            mEdgeEffectBottom.onAbsorb((int)
                    OverScrollerCompat.getCurrVelocity(mScroller));
            mEdgeEffectBottomActive = true;
            needsInvalidate = true;
        }
        ...
    }
```

Here is the section of the code that performs the actual zoom:

```
// Custom object that is functionally similar to Scroller
Zoomer mZoomer;
```

```
private PointF mZoomFocalPoint = new PointF();
...

// If a zoom is in progress (either programmatically or via double
// touch), performs the zoom.
if (mZoomer.computeZoom()) {
    float newWidth = (1f - mZoomer.getCurrZoom()) *
            mScrollerStartViewport.width();
    float newHeight = (1f - mZoomer.getCurrZoom()) *
            mScrollerStartViewport.height();
    float pointWithinViewportX = (mZoomFocalPoint.x -
            mScrollerStartViewport.left)
            / mScrollerStartViewport.width();
    float pointWithinViewportY = (mZoomFocalPoint.y -
            mScrollerStartViewport.top)
            / mScrollerStartViewport.height();
    mCurrentViewport.set(
            mZoomFocalPoint.x - newWidth * pointWithinViewportX,
            mZoomFocalPoint.y - newHeight * pointWithinViewportY,
            mZoomFocalPoint.x + newWidth * (1 -
pointWithinViewportX),
            mZoomFocalPoint.y + newHeight * (1 -
pointWithinViewportY));
    constrainViewport();
    needsInvalidate = true;
}
if (needsInvalidate) {
    ViewCompat.postInvalidateOnAnimation(this);
}
```

This is the `computeScrollSurfaceSize()` method that's called in the above snippet. It computes the current scrollable surface size, in pixels. For example, if the entire chart area is visible, this is simply the current size of `mContentRect`. If the chart is zoomed in 200% in both directions, the returned size will be twice as large horizontally and vertically.

```
private Point computeScrollSurfaceSize() {
    return new Point(
            (int) (mContentRect.width() * (AXIS_X_MAX - AXIS_X_MIN)
                    / mCurrentViewport.width()),
            (int) (mContentRect.height() * (AXIS_Y_MAX - AXIS_Y_MIN)
                    / mCurrentViewport.height()));
}
```

For another example of scroller usage, see the source code for the `ViewPager` class. It scrolls in response to flings, and uses scrolling to implement the "snapping to page" animation.

Next Get started Next class

Handling Multi-Touch Gestures

Previous Next Get started

This lesson teaches you to

1. Track Multiple Pointers
2. Get a MotionEvent's Action

You should also read

- Input Events API Guide
- Sensors Overview
- Making the View Interactive
- Design Guide for Gestures
- Design Guide for Touch Feedback

Try it out

Download the sample

InteractiveChart.zip

A multi-touch gesture is when multiple pointers (fingers) touch the screen at the same time. This lesson describes how to detect gestures that involve multiple pointers.

Track Multiple Pointers

When multiple pointers touch the screen at the same time, the system generates the following touch events:

- `ACTION_DOWN`—For the first pointer that touches the screen. This

starts the gesture. The pointer data for this pointer is always at index 0 in the `MotionEvent`.

- `ACTION_POINTER_DOWN`—For extra pointers that enter the screen beyond the first. The pointer data for this pointer is at the index returned by `getActionIndex()`.
- `ACTION_MOVE`—A change has happened during a press gesture.
- `ACTION_POINTER_UP`—Sent when a non-primary pointer goes up.
- `ACTION_UP`—Sent when the last pointer leaves the screen.

You keep track of individual pointers within a `MotionEvent` via each pointer's index and ID:

- **Index**: A `MotionEvent` effectively stores information about each pointer in an array. The index of a pointer is its position within this array. Most of the `MotionEvent` methods you use to interact with pointers take the pointer index as a parameter, not the pointer ID.
- **ID**: Each pointer also has an ID mapping that stays persistent across touch events to allow tracking an individual pointer across the entire gesture.

The order in which individual pointers appear within a motion event is undefined. Thus the index of a pointer can change from one event to the next, but the pointer ID of a pointer is guaranteed to remain constant as long as the pointer remains active. Use the `getPointerId()` method to obtain a pointer's ID to track the pointer across all subsequent motion events in a gesture. Then for successive motion events, use the `findPointerIndex()` method to obtain the pointer index for a given pointer ID in that motion event. For example:

```
private int mActivePointerId;

public boolean onTouchEvent(MotionEvent event) {
    ....
    // Get the pointer ID
    mActivePointerId = event.getPointerId(0);

    // ... Many touch events later...

    // Use the pointer ID to find the index of the active pointer
    // and fetch its position
    int pointerIndex = event.findPointerIndex(mActivePointerId);
    // Get the pointer's current position
    float x = event.getX(pointerIndex);
    float y = event.getY(pointerIndex);
```

```
}
```

Get a MotionEvent's Action

You should always use the method `getActionMasked()` (or better yet, the compatability version `MotionEventCompat.getActionMasked()`) to retrieve the action of a `MotionEvent`. Unlike the older `getAction()` method, `getActionMasked()` is designed to work with multiple pointers. It returns the masked action being performed, without including the pointer index bits. You can then use `getActionIndex()` to return the index of the pointer associated with the action. This is illustrated in the snippet below.

Note: This example uses the `MotionEventCompat` class. This class is in the Support Library. You should use `MotionEventCompat` to provide the best support for a wide range of platforms. Note that `MotionEventCompat` is *not* a replacement for the `MotionEvent` class. Rather, it provides static utility methods to which you pass your `MotionEvent` object in order to receive the desired action associated with that event.

```
int action = MotionEventCompat.getActionMasked(event);
// Get the index of the pointer associated with the action.
int index = MotionEventCompat.getActionIndex(event);
int xPos = -1;
int yPos = -1;

Log.d(DEBUG_TAG,"The action is " + actionToString(action));

if (event.getPointerCount() > 1) {
    Log.d(DEBUG_TAG,"Multitouch event");
    // The coordinates of the current screen contact, relative to
    // the responding View or Activity.
    xPos = (int)MotionEventCompat.getX(event, index);
    yPos = (int)MotionEventCompat.getY(event, index);

} else {
    // Single touch event
    Log.d(DEBUG_TAG,"Single touch event");
    xPos = (int)MotionEventCompat.getX(event, index);
    yPos = (int)MotionEventCompat.getY(event, index);
}
...

// Given an action int, returns a string description
public static String actionToString(int action) {
    switch (action) {

        case MotionEvent.ACTION_DOWN: return "Down";
        case MotionEvent.ACTION_MOVE: return "Move";
        case MotionEvent.ACTION_POINTER_DOWN: return "Pointer Down";
```

```
        case MotionEvent.ACTION_UP: return "Up";
        case MotionEvent.ACTION_POINTER_UP: return "Pointer Up";
        case MotionEvent.ACTION_OUTSIDE: return "Outside";
        case MotionEvent.ACTION_CANCEL: return "Cancel";
    }
    return "";
}
```

For more discussion of multi-touch and some examples, see the lesson Dragging and Scaling.

Next Get started Next class

Dragging and Scaling

Previous Next Get started

This lesson teaches you to

1. Drag an Object
2. Drag to Pan
3. Use Touch to Perform Scaling

You should also read

- Input Events API Guide
- Sensors Overview
- Making the View Interactive
- Design Guide for Gestures
- Design Guide for Touch Feedback

Try it out

Download the sample

InteractiveChart.zip

This lesson describes how to use touch gestures to drag and scale on-screen objects, using `onTouchEvent()` to intercept touch events.

Drag an Object

If you are targeting Android 3.0 or higher, you can use the built-in drag-and-drop event listeners with `View.OnDragListener`, as described in Drag and Drop.

A common operation for a touch gesture is to use it to drag an object across the screen. The following snippet lets the user drag an on-screen image. Note the following:

- In a drag (or scroll) operation, the app has to keep track of the original pointer (finger), even if additional fingers get placed on the screen. For example, imagine that while dragging the image around, the user places a second finger on the touch screen and lifts the first finger. If your app is just tracking individual pointers, it will regard the second pointer as the default and move the image to that location.
- To prevent this from happening, your app needs to distinguish between the original pointer and any follow-on pointers. To do this, it tracks the `ACTION_POINTER_DOWN` and `ACTION_POINTER_UP` events described in Handling Multi-Touch Gestures. `ACTION_POINTER_DOWN` and `ACTION_POINTER_UP` are passed to the `onTouchEvent()` callback whenever a secondary pointer goes down or up.
- In the `ACTION_POINTER_UP` case, the example extracts this index and ensures that the active pointer ID is not referring to a pointer that is no longer touching the screen. If it is, the app selects a different pointer to be active and saves its current X and Y position. Since this saved position is used in the `ACTION_MOVE` case to calculate the distance to move the onscreen object, the app will always calculate the distance to move using data from the correct pointer.

The following snippet enables a user to drag an object around on the screen. It records the initial position of the active pointer, calculates the distance the pointer traveled, and moves the object to the new position. It correctly manages the possibility of additional pointers, as described above.

Notice that the snippet uses the `getActionMasked()` method. You should always use this method (or better yet, the compatability version `MotionEventCompat.getActionMasked()`) to retrieve the action of a `MotionEvent`. Unlike the older `getAction()` method, `getActionMasked()` is designed to work with multiple pointers. It returns the masked action being performed, without including the pointer index bits.

```
// The ‘active pointer’ is the one currently moving our object.
private int mActivePointerId = INVALID_POINTER_ID;

@Override
public boolean onTouchEvent(MotionEvent ev) {
    // Let the ScaleGestureDetector inspect all events.
    mScaleDetector.onTouchEvent(ev);

    final int action = MotionEventCompat.getActionMasked(ev);

    switch (action) {
    case MotionEvent.ACTION_DOWN: {
        final int pointerIndex =
MotionEventCompat.getActionIndex(ev);
        final float x = MotionEventCompat.getX(ev, pointerIndex);
        final float y = MotionEventCompat.getY(ev, pointerIndex);

        // Remember where we started (for dragging)
        mLastTouchX = x;
        mLastTouchY = y;
        // Save the ID of this pointer (for dragging)
        mActivePointerId = MotionEventCompat.getPointerId(ev, 0);
        break;
    }

    case MotionEvent.ACTION_MOVE: {
        // Find the index of the active pointer and fetch its
position
        final int pointerIndex =
                MotionEventCompat.findPointerIndex(ev,
mActivePointerId);

        final float x = MotionEventCompat.getX(ev, pointerIndex);
        final float y = MotionEventCompat.getY(ev, pointerIndex);

        // Calculate the distance moved
        final float dx = x - mLastTouchX;
        final float dy = y - mLastTouchY;

        mPosX += dx;
        mPosY += dy;

        invalidate();

        // Remember this touch position for the next move event
        mLastTouchX = x;
        mLastTouchY = y;

        break;
    }

    case MotionEvent.ACTION_UP: {
        mActivePointerId = INVALID_POINTER_ID;
        break;
    }

    case MotionEvent.ACTION_CANCEL: {
        mActivePointerId = INVALID_POINTER_ID;
        break;
    }
```

```
    case MotionEvent.ACTION_POINTER_UP: {

        final int pointerIndex =
MotionEventCompat.getActionIndex(ev);
        final int pointerId = MotionEventCompat.getPointerId(ev,
pointerIndex);

        if (pointerId == mActivePointerId) {
            // This was our active pointer going up. Choose a new
            // active pointer and adjust accordingly.
            final int newPointerIndex = pointerIndex == 0 ? 1 : 0;
            mLastTouchX = MotionEventCompat.getX(ev,
newPointerIndex);
            mLastTouchY = MotionEventCompat.getY(ev,
newPointerIndex);
            mActivePointerId = MotionEventCompat.getPointerId(ev,
newPointerIndex);
        }
        break;
    }
    }
    return true;
}
```

Drag to Pan

The previous section showed an example of dragging an object around the screen. Another common scenario is *panning*, which is when a user's dragging motion causes scrolling in both the x and y axes. The above snippet directly intercepted the `MotionEvent` actions to implement dragging. The snippet in this section takes advantage of the platform's built-in support for common gestures. It overrides `onScroll()` in `GestureDetector.SimpleOnGestureListener`.

To provide a little more context, `onScroll()` is called when a user is dragging his finger to pan the content. `onScroll()` is only called when a finger is down; as soon as the finger is lifted from the screen, the gesture either ends, or a fling gesture is started (if the finger was moving with some speed just before it was lifted). For more discussion of scrolling vs. flinging, see Animating a Scroll Gesture.

Here is the snippet for `onScroll()`:

```
// The current viewport. This rectangle represents the currently
visible
// chart domain and range.
private RectF mCurrentViewport =
        new RectF(AXIS_X_MIN, AXIS_Y_MIN, AXIS_X_MAX, AXIS_Y_MAX);
```

```
// The current destination rectangle (in pixel coordinates) into
which the
// chart data should be drawn.
private Rect mContentRect;

private final GestureDetector.SimpleOnGestureListener
mGestureListener
            = new GestureDetector.SimpleOnGestureListener() {
...

@Override
public boolean onScroll(MotionEvent e1, MotionEvent e2,
            float distanceX, float distanceY) {
    // Scrolling uses math based on the viewport (as opposed to math
using pixels).

    // Pixel offset is the offset in screen pixels, while viewport
offset is the
    // offset within the current viewport.
    float viewportOffsetX = distanceX * mCurrentViewport.width()
            / mContentRect.width();
    float viewportOffsetY = -distanceY * mCurrentViewport.height()
            / mContentRect.height();
    ...
    // Updates the viewport, refreshes the display.
    setViewportBottomLeft(
            mCurrentViewport.left + viewportOffsetX,
            mCurrentViewport.bottom + viewportOffsetY);
    ...
    return true;
}
```

The implementation of `onScroll()` scrolls the viewport in response to the touch gesture:

```
/**
 * Sets the current viewport (defined by mCurrentViewport) to the
given
 * X and Y positions. Note that the Y value represents the topmost
pixel position,
 * and thus the bottom of the mCurrentViewport rectangle.
 */
private void setViewportBottomLeft(float x, float y) {
    /*
     * Constrains within the scroll range. The scroll range is
simply the viewport
     * extremes (AXIS_X_MAX, etc.) minus the viewport size. For
example, if the
     * extremes were 0 and 10, and the viewport size was 2, the
scroll range would
     * be 0 to 8.
     */

    float curWidth = mCurrentViewport.width();
    float curHeight = mCurrentViewport.height();
    x = Math.max(AXIS_X_MIN, Math.min(x, AXIS_X_MAX - curWidth));
    y = Math.max(AXIS_Y_MIN + curHeight, Math.min(y, AXIS_Y_MAX));
```

```
    mCurrentViewport.set(x, y - curHeight, x + curWidth, y);

    // Invalidates the View to update the display.
    ViewCompat.postInvalidateOnAnimation(this);
}
```

Use Touch to Perform Scaling

As discussed in Detecting Common Gestures, `GestureDetector` helps you detect common gestures used by Android such as scrolling, flinging, and long press. For scaling, Android provides `ScaleGestureDetector`. `GestureDetector` and `ScaleGestureDetector` can be used together when you want a view to recognize additional gestures.

To report detected gesture events, gesture detectors use listener objects passed to their constructors. `ScaleGestureDetector` uses `ScaleGestureDetector.OnScaleGestureListener`. Android provides `ScaleGestureDetector.SimpleOnScaleGestureListener` as a helper class that you can extend if you don't care about all of the reported events.

Basic scaling example

Here is a snippet that illustrates the basic ingredients involved in scaling.

```
private ScaleGestureDetector mScaleDetector;
private float mScaleFactor = 1.f;

public MyCustomView(Context mContext){
    ...
    // View code goes here
    ...
    mScaleDetector = new ScaleGestureDetector(context, new
ScaleListener());
}

@Override
public boolean onTouchEvent(MotionEvent ev) {
    // Let the ScaleGestureDetector inspect all events.
    mScaleDetector.onTouchEvent(ev);
    return true;
}

@Override
public void onDraw(Canvas canvas) {
    super.onDraw(canvas);

    canvas.save();
    canvas.scale(mScaleFactor, mScaleFactor);
    ...
```

```
    // onDraw() code goes here
    ...
    canvas.restore();
}

private class ScaleListener
        extends ScaleGestureDetector.SimpleOnScaleGestureListener {
    @Override
    public boolean onScale(ScaleGestureDetector detector) {
        mScaleFactor *= detector.getScaleFactor();

        // Don't let the object get too small or too large.
        mScaleFactor = Math.max(0.1f, Math.min(mScaleFactor, 5.0f));

        invalidate();
        return true;
    }
}
```

More complex scaling example

Here is a more complex example from the `InteractiveChart` sample provided with this class. The `InteractiveChart` sample supports both scrolling (panning) and scaling with multiple fingers, using the `ScaleGestureDetector` "span" (`getCurrentSpanX/Y`) and "focus" (`getFocusX/Y`) features:

```
@Override
private RectF mCurrentViewport =
        new RectF(AXIS_X_MIN, AXIS_Y_MIN, AXIS_X_MAX, AXIS_Y_MAX);
private Rect mContentRect;
private ScaleGestureDetector mScaleGestureDetector;
...
public boolean onTouchEvent(MotionEvent event) {
    boolean retVal = mScaleGestureDetector.onTouchEvent(event);
    retVal = mGestureDetector.onTouchEvent(event) || retVal;
    return retVal || super.onTouchEvent(event);
}

/**
 * The scale listener, used for handling multi-finger scale
gestures.
 */
private final ScaleGestureDetector.OnScaleGestureListener
mScaleGestureListener
        = new ScaleGestureDetector.SimpleOnScaleGestureListener() {
    /**
     * This is the active focal point in terms of the viewport.
Could be a local
     * variable but kept here to minimize per-frame allocations.
     */
    private PointF viewportFocus = new PointF();
    private float lastSpanX;
    private float lastSpanY;
```

```
    // Detects that new pointers are going down.
    @Override
    public boolean onScaleBegin(ScaleGestureDetector
scaleGestureDetector) {
        lastSpanX = ScaleGestureDetectorCompat.
                getCurrentSpanX(scaleGestureDetector);
        lastSpanY = ScaleGestureDetectorCompat.
                getCurrentSpanY(scaleGestureDetector);
        return true;
    }

    @Override
    public boolean onScale(ScaleGestureDetector scaleGestureDetector)
{

        float spanX = ScaleGestureDetectorCompat.
                getCurrentSpanX(scaleGestureDetector);
        float spanY = ScaleGestureDetectorCompat.
                getCurrentSpanY(scaleGestureDetector);

        float newWidth = lastSpanX / spanX *
mCurrentViewport.width();
        float newHeight = lastSpanY / spanY *
mCurrentViewport.height();

        float focusX = scaleGestureDetector.getFocusX();
        float focusY = scaleGestureDetector.getFocusY();
        // Makes sure that the chart point is within the chart
region.
        // See the sample for the implementation of hitTest().
        hitTest(scaleGestureDetector.getFocusX(),
                scaleGestureDetector.getFocusY(),
                viewportFocus);

        mCurrentViewport.set(
                viewportFocus.x
                        - newWidth * (focusX - mContentRect.left)
                        / mContentRect.width(),
                viewportFocus.y
                        - newHeight * (mContentRect.bottom - focusY)
                        / mContentRect.height(),
                0,
                0);
        mCurrentViewport.right = mCurrentViewport.left + newWidth;
        mCurrentViewport.bottom = mCurrentViewport.top + newHeight;
        ...
        // Invalidates the View to update the display.

ViewCompat.postInvalidateOnAnimation(InteractiveLineGraphView.this);

        lastSpanX = spanX;
        lastSpanY = spanY;
        return true;
    }
};
```

Next Get started Next class

Managing Touch Events in a ViewGroup

Previous Next Get started

This lesson teaches you to

1. Intercept Touch Events in a ViewGroup
2. Use ViewConfiguration Constants
3. Extend a Child View's Touchable Area

You should also read

- Input Events API Guide
- Sensors Overview
- Making the View Interactive
- Design Guide for Gestures
- Design Guide for Touch Feedback

Try it out

Download the sample

InteractiveChart.zip

Handling touch events in a `ViewGroup` takes special care, because it's common for a `ViewGroup` to have children that are targets for different touch events than the `ViewGroup` itself. To make sure that each view correctly receives the touch events intended for it, override the `onInterceptTouchEvent()` method.

Intercept Touch Events in a ViewGroup

The `onInterceptTouchEvent()` method is called whenever a touch event is detected on the surface of a `ViewGroup`, including on the surface of its children. If `onInterceptTouchEvent()` returns `true`, the `MotionEvent` is intercepted, meaning it will be not be passed on to the child, but rather to the `onTouchEvent()` method of the parent.

The `onInterceptTouchEvent()` method gives a parent the chance to see any touch event before its children do. If you return `true` from `onInterceptTouchEvent()`, the child view that was previously handling touch events receives an `ACTION_CANCEL`, and the events from that point forward are sent to the parent's `onTouchEvent()` method for the usual handling. `onInterceptTouchEvent()` can also return `false` and simply spy on events as they travel down the view hierarchy to their usual targets, which will handle the events with their own `onTouchEvent()`.

In the following snippet, the class `MyViewGroup` extends `ViewGroup`. `MyViewGroup` contains multiple child views. If you drag your finger across a child view horizontally, the child view should no longer get touch events, and `MyViewGroup` should handle touch events by scrolling its contents. However, if you press buttons in the child view, or scroll the child view vertically, the parent shouldn't intercept those touch events, because the child is the intended target. In those cases, `onInterceptTouchEvent()` should return `false`, and `MyViewGroup`'s `onTouchEvent()` won't be called.

```
public class MyViewGroup extends ViewGroup {

    private int mTouchSlop;

    ...

    ViewConfiguration vc = ViewConfiguration.get(view.getContext());
    mTouchSlop = vc.getScaledTouchSlop();

    ...

    @Override
    public boolean onInterceptTouchEvent(MotionEvent ev) {
        /*
         * This method JUST determines whether we want to intercept
the motion.
         * If we return true, onTouchEvent will be called and we do
the actual
         * scrolling there.
```

```
         */


        final int action = MotionEventCompat.getActionMasked(ev);

        // Always handle the case of the touch gesture being
complete.
        if (action == MotionEvent.ACTION_CANCEL || action ==
MotionEvent.ACTION_UP) {
            // Release the scroll.
            mIsScrolling = false;
            return false; // Do not intercept touch event, let the
child handle it
        }

        switch (action) {
            case MotionEvent.ACTION_MOVE: {
                if (mIsScrolling) {
                    // We're currently scrolling, so yes, intercept
the
                    // touch event!
                    return true;
                }

                // If the user has dragged her finger horizontally
more than
                // the touch slop, start the scroll

                // left as an exercise for the reader
                final int xDiff = calculateDistanceX(ev);

                // Touch slop should be calculated using
ViewConfiguration
                // constants.
                if (xDiff > mTouchSlop) {
                    // Start scrolling!
                    mIsScrolling = true;
                    return true;
                }
                break;
            }
            ...
        }

        // In general, we don't want to intercept touch events. They
should be
        // handled by the child view.
        return false;
    }

    @Override
    public boolean onTouchEvent(MotionEvent ev) {
        // Here we actually handle the touch event (e.g. if the
action is ACTION_MOVE,
        // scroll this container).
        // This method will only be called if the touch event was
intercepted in
        // onInterceptTouchEvent
        ...
    }
}
```

Note that `ViewGroup` also provides a `requestDisallowInterceptTouchEvent()` method. The `ViewGroup` calls this method when a child does not want the parent and its ancestors to intercept touch events with `onInterceptTouchEvent()`.

Use ViewConfiguration Constants

The above snippet uses the current `ViewConfiguration` to initialize a variable called `mTouchSlop`. You can use the `ViewConfiguration` class to access common distances, speeds, and times used by the Android system.

"Touch slop" refers to the distance in pixels a user's touch can wander before the gesture is interpreted as scrolling. Touch slop is typically used to prevent accidental scrolling when the user is performing some other touch operation, such as touching on-screen elements.

Two other commonly used `ViewConfiguration` methods are `getScaledMinimumFlingVelocity()` and `getScaledMaximumFlingVelocity()`. These methods return the minimum and maximum velocity (respectively) to initiate a fling, as measured in pixels per second. For example:

```
ViewConfiguration vc = ViewConfiguration.get(view.getContext());
private int mSlop = vc.getScaledTouchSlop();
private int mMinFlingVelocity = vc.getScaledMinimumFlingVelocity();
private int mMaxFlingVelocity = vc.getScaledMaximumFlingVelocity();

...

case MotionEvent.ACTION_MOVE: {
    ...
    float deltaX = motionEvent.getRawX() - mDownX;
    if (Math.abs(deltaX) > mSlop) {
        // A swipe occurred, do something
    }

...

case MotionEvent.ACTION_UP: {
    ...
    } if (mMinFlingVelocity <= velocityX && velocityX <=
mMaxFlingVelocity
            && velocityY < velocityX) {
        // The criteria have been satisfied, do something
    }
}
```

Extend a Child View's Touchable Area

Android provides the `TouchDelegate` class to make it possible for a parent to extend the touchable area of a child view beyond the child's bounds. This is useful when the child has to be small, but should have a larger touch region. You can also use this approach to shrink the child's touch region if need be.

In the following example, an `ImageButton` is the "delegate view" (that is, the child whose touch area the parent will extend). Here is the layout file:

```
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
     android:id="@+id/parent_layout"
     android:layout_width="match_parent"
     android:layout_height="match_parent"
     tools:context=".MainActivity" >

     <ImageButton android:id="@+id/button"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:background="@null"
          android:src="@drawable/icon" />
</RelativeLayout>
```

The snippet below does the following:

- Gets the parent view and posts a `Runnable` on the UI thread. This ensures that the parent lays out its children before calling the `getHitRect()` method. The `getHitRect()` method gets the child's hit rectangle (touchable area) in the parent's coordinates.
- Finds the `ImageButton` child view and calls `getHitRect()` to get the bounds of the child's touchable area.
- Extends the bounds of the `ImageButton`'s hit rectangle.
- Instantiates a `TouchDelegate`, passing in the expanded hit rectangle and the `ImageButton` child view as parameters.
- Sets the `TouchDelegate` on the parent view, such that touches within the touch delegate bounds are routed to the child.

In its capacity as touch delegate for the `ImageButton` child view, the parent view will receive all touch events. If the touch event occurred within the child's hit rectangle, the parent will pass the touch event to

the child for handling.

```
public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // Get the parent view
        View parentView = findViewById(R.id.parent_layout);

        parentView.post(new Runnable() {
            // Post in the parent's message queue to make sure the
parent
            // lays out its children before you call getHitRect()
            @Override
            public void run() {
                // The bounds for the delegate view (an ImageButton
                // in this example)
                Rect delegateArea = new Rect();
                ImageButton myButton = (ImageButton)
findViewById(R.id.button);
                myButton.setEnabled(true);
                myButton.setOnClickListener(new
View.OnClickListener() {
                    @Override
                    public void onClick(View view) {
                        Toast.makeText(MainActivity.this,
                                "Touch occurred within ImageButton
touch region.",
                                Toast.LENGTH_SHORT).show();
                    }
                });

                // The hit rectangle for the ImageButton
                myButton.getHitRect(delegateArea);

                // Extend the touch area of the ImageButton beyond
its bounds
                // on the right and bottom.
                delegateArea.right += 100;
                delegateArea.bottom += 100;

                // Instantiate a TouchDelegate.
                // "delegateArea" is the bounds in local coordinates
of
                // the containing view to be mapped to the delegate
view.
                // "myButton" is the child view that should receive
motion
                // events.
                TouchDelegate touchDelegate = new
TouchDelegate(delegateArea,
                        myButton);

                // Sets the TouchDelegate on the parent view, such
that touches
                // within the touch delegate bounds are routed to
the child.
                if (View.class.isInstance(myButton.getParent())) {
```

```
                    ((View)
myButton.getParent()).setTouchDelegate(touchDelegate);
                }
            }
        });
    }
}
```

Next Get started Next class

Handling Keyboard Input

Previous Next Get started

Dependencies and prerequisites

- Android 1.6 (API Level 3) or higher

The Android system shows an on-screen keyboard—known as a *soft input method*—when a text field in your UI receives focus. To provide the best user experience, you can specify characteristics about the type of input you expect (such as whether it's a phone number or email address) and how the input method should behave (such as whether it performs auto-correct for spelling mistakes).

In addition to the on-screen input methods, Android also supports hardware keyboards, so it's important that your app optimize its user experience for interaction that might occur through an attached keyboard.

These topics and more are discussed in the following lessons.

Lessons

Specifying the Input Method Type
: Learn how to show certain soft input methods, such as those designed for phone numbers, web addresses, or other formats. Also learn how to specify characteristics such as spelling suggestion behavior and action buttons such as **Done** or **Next**.

Handling Input Method Visibility
: Learn how to specify when to show the soft input method and how your layout should adjust to the reduced screen space.

Supporting Keyboard Navigation

Learn how to verify that users can navigate your app using a keyboard and how to make any necessary changes to the navigation order.

Handling Keyboard Actions

Learn how to respond directly to keyboard input for user actions.

Next Get started Next class

Specifying the Input Method Type

Previous Next Get started

This lesson teaches you to

1. Specify the Keyboard Type
2. Enable Spelling Suggestions and Other Behaviors
3. Specify the Input Method Action

You should also read

- Text Fields

Every text field expects a certain type of text input, such as an email address, phone number, or just plain text. So it's important that you specify the input type for each text field in your app so the system displays the appropriate soft input method (such as an on-screen keyboard).

Beyond the type of buttons available with an input method, you should specify behaviors such as whether the input method provides spelling suggestions, capitalizes new sentences, and replaces the carriage return button with an action button such as a **Done** or **Next**. This lesson shows how to specify these characteristics.

Specify the Keyboard Type

You should always declare the input method for your text fields by adding the `android:inputType` attribute to the `<EditText>` element.

Figure 1. The `phone` input type.

For example, if you'd like an input method for entering a phone number, use the `"phone"` value:

```
<EditText
    android:id="@+id/phone"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:hint="@string/phone_hint"
    android:inputType="phone" />
```

Figure 2. The `textPassword` input type.

Or if the text field is for a password, use the `"textPassword"` value so the text field conceals the user's input:

```
<EditText
    android:id="@+id/password"
    android:hint="@string/password_hint"
    android:inputType="textPassword"
    ... />
```

There are several possible values documented with the `android:inputType` attribute and some of the values can be combined to specify the input method appearance and additional behaviors.

Enable Spelling Suggestions and Other Behaviors

Figure 3. Adding `textAutoCorrect` provides auto-correction for misspellings.

The `android:inputType` attribute allows you to specify various behaviors for the input method. Most importantly, if your text field is intended for basic text input (such as for a text message), you should enable auto spelling correction with the `"textAutoCorrect"` value.

You can combine different behaviors and input method styles with the `android:inputType` attribute. For example, here's how to create a text field that capitalizes the first word of a sentence and also auto-corrects misspellings:

```
<EditText
    android:id="@+id/message"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:inputType=
```

```
        "textCapSentences|textAutoCorrect"
    ... />
```

Specify the Input Method Action

Most soft input methods provide a user action button in the bottom corner that's appropriate for the current text field. By default, the system uses this button for either a **Next** or **Done** action unless your text field allows multi-line text (such as with `android:inputType="textMultiLine"`), in which case the action button is a carriage return. However, you can specify additional actions that might be more appropriate for your text field, such as **Send** or **Go**.

To specify the keyboard action button, use the `android:imeOptions` attribute with an action value such as `"actionSend"` or `"actionSearch"`. For example:

Figure 4. The Send button appears when you declare `android:imeOptions="actionSend"`.

```
<EditText
    android:id="@+id/search"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:hint="@string/search_hint"
    android:inputType="text"
    android:imeOptions="actionSend" />
```

You can then listen for presses on the action button by defining a `TextView.OnEditorActionListener` for the `EditText` element. In your listener, respond to the appropriate IME action ID defined in the `EditorInfo` class, such as `IME_ACTION_SEND`. For example:

```
EditText editText = (EditText) findViewById(R.id.search);
editText.setOnEditorActionListener(new OnEditorActionListener() {
    @Override
    public boolean onEditorAction(TextView v, int actionId, KeyEvent
event) {
        boolean handled = false;
```

```
        if (actionId == EditorInfo.IME_ACTION_SEND) {
            sendMessage();
            handled = true;
        }
        return handled;
    }
});
```

Next Get started Next class

Handling Input Method Visibility

Previous Next Get started

This lesson teaches you to

1. Show the Input Method When the Activity Starts
2. Show the Input Method On Demand
3. Specify How Your UI Should Respond

When input focus moves into or out of an editable text field, Android shows or hides the input method (such as the on-screen keyboard) as appropriate. The system also makes decisions about how your UI and the text field appear above the input method. For example, when the vertical space on the screen is constrained, the text field might fill all space above the input method. For most apps, these default behaviors are all that's needed.

In some cases, though, you might want to more directly control the visibility of the input method and specify how you'd like your layout to appear when the input method is visible. This lesson explains how to control and respond to the input method visibility.

Show the Input Method When the Activity Starts

Although Android gives focus to the first text field in your layout when the activity starts, it does not show the input method. This behavior is appropriate because entering text might not be the primary task in the activity. However, if entering text is indeed the primary task (such as in a login screen), then you probably want the input method to appear by default.

To show the input method when your activity starts, add the `android:windowSoftInputMode` attribute to the `<activity>` element with the `"stateVisible"` value. For example:

```
<application ... >
    <activity
        android:windowSoftInputMode="stateVisible" ... >
        ...
    </activity>
    ...
</application>
```

Note: If the user's device has an attached hardware keyboard, the soft input method *does not* appear.

Show the Input Method On Demand

If there is a method in your activity's lifecycle where you want to ensure that the input method is visible, you can use the `InputMethodManager` to show it.

For example, the following method takes a `View` in which the user should type something, calls `requestFocus()` to give it focus, then `showSoftInput()` to open the input method:

```
public void showSoftKeyboard(View view) {
    if (view.requestFocus()) {
        InputMethodManager imm = (InputMethodManager)
                getSystemService(Context.INPUT_METHOD_SERVICE);
        imm.showSoftInput(view, InputMethodManager.SHOW_IMPLICIT);
    }
}
```

Note: Once the input method is visible, you should not programmatically hide it. The system hides the input method when the user finishes the task in the text field or the user can hide it with a system control (such as with the *Back* button).

Specify How Your UI Should Respond

When the input method appears on the screen, it reduces the amount of space available for your app's UI. The system makes a decision as to how it should adjust the visible portion of your UI, but it might not get it right.

To ensure the best behavior for your app, you should specify how you'd like the system to display your UI in the remaining space.

To declare your preferred treatment in an activity, use the `android:windowSoftInputMode` attribute in your manifest's `<activity>` element with one of the "adjust" values.

For example, to ensure that the system resizes your layout to the available space—which ensures that all of your layout content is accessible (even though it probably requires scrolling)—use `"adjustResize"`:

```
<application ... >
    <activity
        android:windowSoftInputMode="adjustResize" ... >
        ...
    </activity>
    ...
</application>
```

You can combine the adjustment specification with the initial input method visibility specification from above:

```
    <activity
        android:windowSoftInputMode="stateVisible|adjustResize" ... >
        ...
    </activity>
```

Specifying `"adjustResize"` is important if your UI includes controls that the user might need to access immediately after or while performing text input. For example, if you use a relative layout to place a button bar at the bottom of the screen, using `"adjustResize"` resizes the layout so the button bar appears above the input method.

Next Get started Next class

Supporting Keyboard Navigation

Previous Next Get started

This lesson teaches you to

1. Test Your App
2. Handle Tab Navigation
3. Handle Directional Navigation

You should also read

- Implementing Accessibility

In addition to soft input methods (such as on-screen keyboards), Android supports physical keyboards attached to the device. A keyboard offers not only a convenient mode for text input, but also offers a way for users to navigate and interact with your app. Although most hand-held devices such as phones use touch as the primary mode of interaction, tablets and similar devices are growing in popularity and many users like to attach keyboard accessories.

As more Android devices offer this kind of experience, it's important that you optimize your app to support interaction through a keyboard. This lesson describes how you can better support navigation with a keyboard.

Note: Supporting of directional navigation in your application is also important in ensuring that your application is accessible to users who do not navigate using visual cues. Fully supporting directional navigation in your application can also help you automate user interface testing with tools like uiautomator.

Test Your App

It's possible that users can already navigate your app using a keyboard, because the Android system enables most of the necessary behaviors by default.

All interactive widgets provided by the Android framework (such as `Button` and `EditText`) are focusable. This means users can navigate with control devices such as a D-pad or keyboard and each widget glows or otherwise changes its appearance when it gains input focus.

To test your app:

1. Install your app on a device that offers a hardware keyboard.

   If you don't have a hardware device with a keyboard, connect a Bluetooth keyboard or a USB keyboard (though not all devices support USB accessories).

   You can also use the Android emulator:

   1. In the AVD Manager, either click **New Device** or select an existing profile and click **Clone**.
   2. In the window that appears, ensure that **Keyboard** and **DPad** are enabled.
2. To test your app, use only the Tab key to navigate through your UI, ensuring that each UI control gets focus as expected.

   Look for any instances in which the focus moves in a way you don't expect.

3. Start from the beginning of your app and instead use the direction controls (arrow keys on the keyboard) to navigate your app.

   From each focusable element in your UI, press Up, Down, Left, and Right.

   Look for any instances in which the focus moves in a way you don't expect.

If you encounter any instances where navigating with the Tab key or

direction controls does not do what you expect, specify where the focus should go in your layout, as discussed in the following sections.

Handle Tab Navigation

When a user navigates your app using the keyboard Tab key, the system passes input focus between elements based on the order in which they appear in the layout. If you use a relative layout, for example, and the order of elements on the screen is different than the order in the file, then you might need to manually specify the focus order.

For example, in the following layout, two buttons are aligned to the right side and a text field is aligned to the left of the second button. In order to pass focus from the first button to the text field, then to the second button, the layout needs to explicitly define the focus order for each of the focusable elements with the `android:nextFocusForward` attribute:

```
<RelativeLayout ...>
    <Button
        android:id="@+id/button1"
        android:layout_alignParentTop="true"
        android:layout_alignParentRight="true"
        android:nextFocusForward="@+id/editText1"
        ... />
    <Button
        android:id="@+id/button2"
        android:layout_below="@id/button1"
        android:nextFocusForward="@+id/button1"
        ... />
    <EditText
        android:id="@id/editText1"
        android:layout_alignBottom="@+id/button2"
        android:layout_toLeftOf="@id/button2"
        android:nextFocusForward="@+id/button2"
        ...  />
    ...
</RelativeLayout>
```

Now instead of sending focus from `button1` to `button2` then `editText1`, the focus appropriately moves according to the appearance on the screen: from `button1` to `editText1` then `button2`.

Handle Directional Navigation

Users can also navigate your app using the arrow keys on a keyboard

(the behavior is the same as when navigating with a D-pad or trackball). The system provides a best-guess as to which view should be given focus in a given direction based on the layout of the views on screen. Sometimes, however, the system might guess wrong.

If the system does not pass focus to the appropriate view when navigating in a given direction, specify which view should receive focus with the following attributes:

- `android:nextFocusUp`
- `android:nextFocusDown`
- `android:nextFocusLeft`
- `android:nextFocusRight`

Each attribute designates the next view to receive focus when the user navigates in that direction, as specified by the view ID. For example:

```
<Button
    android:id="@+id/button1"
    android:nextFocusRight="@+id/button2"
    android:nextFocusDown="@+id/editText1"
    ... />
<Button
    android:id="@id/button2"
    android:nextFocusLeft="@id/button1"
    android:nextFocusDown="@id/editText1"
    ... />
<EditText
    android:id="@id/editText1"
    android:nextFocusUp="@id/button1"
    ...  />
```

Next Get started Next class

Handling Keyboard Actions

Previous Next Get started

This lesson teaches you to

1. Handle Single Key Events
2. Handle Modifier Keys

When the user gives focus to an editable text view such as an `EditText` element and the user has a hardware keyboard attached, all input is handled by the system. If, however, you'd like to intercept or directly handle the keyboard input yourself, you can do so by implementing callback methods from the `KeyEvent.Callback` interface, such as `onKeyDown()` and `onKeyMultiple()`.

Both the `Activity` and `View` class implement the `KeyEvent.Callback` interface, so you should generally override the callback methods in your extension of these classes as appropriate.

Note: When handling keyboard events with the `KeyEvent` class and related APIs, you should expect that such keyboard events come only from a hardware keyboard. You should never rely on receiving key events for any key on a soft input method (an on-screen keyboard).

Handle Single Key Events

To handle an individual key press, implement `onKeyDown()` or `onKeyUp()` as appropriate. Usually, you should use `onKeyUp()` if you want to be sure that you receive only one event. If the user presses and holds the button, then `onKeyDown()` is called multiple times.

For example, this implementation responds to some keyboard keys to control a game:

```
@Override
public boolean onKeyUp(int keyCode, KeyEvent event) {
    switch (keyCode) {
        case KeyEvent.KEYCODE_D:
            moveShip(MOVE_LEFT);
            return true;
        case KeyEvent.KEYCODE_F:
            moveShip(MOVE_RIGHT);
            return true;
        case KeyEvent.KEYCODE_J:
            fireMachineGun();
            return true;
        case KeyEvent.KEYCODE_K:
            fireMissile();
            return true;
        default:
            return super.onKeyUp(keyCode, event);
    }
}
```

Handle Modifier Keys

To respond to modifier key events such as when a key is combined with Shift or Control, you can query the `KeyEvent` that's passed to the callback method. Several methods provide information about modifier keys such as `getModifiers()` and `getMetaState()`. However, the simplest solution is to check whether the exact modifier key you care about is being pressed with methods such as `isShiftPressed()` and `isCtrlPressed()`.

For example, here's the `onKeyDown()` implementation again, with some extra handling for when the Shift key is held down with one of the keys:

```
@Override
public boolean onKeyUp(int keyCode, KeyEvent event) {
    switch (keyCode) {
        ...
        case KeyEvent.KEYCODE_J:
            if (event.isShiftPressed()) {
                fireLaser();
            } else {
                fireMachineGun();
            }
            return true;
        case KeyEvent.KEYCODE_K:
            if (event.isShiftPressed()) {
                fireSeekingMissle();
            } else {
                fireMissile();
            }
            return true;
        default:
```

```
            return super.onKeyUp(keyCode, event);
    }
}
```

Next Get started Next class

Supporting Game Controllers

Previous Next Get started

Dependencies and prerequisites

- Android 2.3 (API level 9) or higher.

You should also read

- Key Layout Files
- Input Events

Try it out

Download the sample

ControllerSample.zip

You can greatly enhance the user experience in your game by letting players use their favorite game controllers. The Android framework provides APIs for detecting and processing user input from game controllers.

This class shows how to make your game work consistently with game controllers across different Android API levels (API level 9 and up), and how to enhance the gaming experience for players by supporting multiple controllers simultaneously in your app.

Lessons

Handling Controller Actions

Learn how to handle user input from common input elements on

game controllers, including directional pad (D-pad) buttons, gamepad buttons, and joysticks.

Supporting Controllers Across Android Versions

Learn how to make game controllers behave the same across devices running different versions of Android.

Supporting Multiple Game Controllers

Learn how to detect and use multiple game controllers that are simultaneously connected.

Next Get started Next class

Handling Controller Actions

Previous Next Get started

This lesson teaches you to

1. Verify a Game Controller is Connected
2. Process Gamepad Button Presses
3. Process Directional Pad Input
4. Process Joystick Movements

Try it out

Download the sample

ControllerSample.zip

At the system level, Android reports input event codes from game controllers as Android key codes and axis values. In your game, you can receive these codes and values and convert them to specific in-game actions.

When players physically connect or wirelessly pair a game controller to their Android-powered devices, the system auto-detects the controller as an input device and starts reporting its input events. Your game can receive these input events by implementing the following callback methods in your active `Activity` or focused `View` (you should implement the callbacks for either the `Activity` or `View`, but not both):

- From `Activity`:

- `dispatchGenericMotionEvent(android.view. MotionEvent)`

  Called to process generic motion events such as joystick movements.

- `dispatchKeyEvent(android.view.KeyEvent)`

  Called to process key events such as a press or release of a gamepad or D-pad button.

- From `View`:
  - `onGenericMotionEvent(android.view.MotionEvent)`

    Called to process generic motion events such as joystick movements.

  - `onKeyDown(int, android.view.KeyEvent)`

    Called to process a press of a physical key such as a gamepad or D-pad button.

  - `onKeyUp(int, android.view.KeyEvent)`

    Called to process a release of a physical key such as a gamepad or D-pad button.

The recommended approach is to capture the events from the specific `View` object that the user interacts with. Inspect the following objects provided by the callbacks to get information about the type of input event received:

`KeyEvent`
: An object that describes directional pad (D-pad) and gamepad button events. Key events are accompanied by a *key code* that indicates the specific button triggered, such as `DPAD_DOWN` or `BUTTON_A`. You can obtain the key code by calling `getKeyCode()` or from key event callbacks such as `onKeyDown()`.

`MotionEvent`
: An object that describes input from joystick and shoulder trigger movements. Motion events are accompanied by an action code and a set of *axis values*. The action code specifies the state change that occurred such as a joystick

being moved. The axis values describe the position and other movement properties for a specific physical control, such as `AXIS_X` or `AXIS_RTRIGGER`. You can obtain the action code by calling `getAction()` and the axis value by calling `getAxisValue()`.

This lesson focuses on how you can handle input from the most common types of physical controls (gamepad buttons, directional pads, and joysticks) in a game screen by implementing the above-mentioned `View` callback methods and processing `KeyEvent` and `MotionEvent` objects.

Verify a Game Controller is Connected

When reporting input events, Android does not distinguish between events that came from a non-game controller device and events that came from a game controller. For example, a touch screen action generates an `AXIS_X` event that represents the X coordinate of the touch surface, but a joystick generates an `AXIS_X` event that represents the X position of the joystick. If your game cares about handling game-controller input, you should first check that the input event comes from a relevant source type.

To verify that a connected input device is a game controller, call `getSources()` to obtain a combined bit field of input source types supported on that device. You can then test to see if the following fields are set:

- A source type of `SOURCE_GAMEPAD` indicates that the input device has gamepad buttons (for example, `BUTTON_A`). Note that this source type does not strictly indicate if the game controller has D-pad buttons, although most gamepads typically have directional controls.
- A source type of `SOURCE_DPAD` indicates that the input device has D-pad buttons (for example, `DPAD_UP`).
- A source type of `SOURCE_JOYSTICK` indicates that the input device has analog control sticks (for example, a joystick that records movements along `AXIS_X` and `AXIS_Y`).

The following code snippet shows a helper method that lets you check whether the connected input devices are game controllers. If so, the method retrieves the device IDs for the game controllers. You can then associate each device ID with a player in your game, and process game actions for each connected player separately. To learn more about supporting multiple game controllers that are simultaneously connected on the same Android device, see Supporting

Multiple Game Controllers.

```
public ArrayList getGameControllerIds() {
    ArrayList gameControllerDeviceIds = new ArrayList();
    int[] deviceIds = InputDevice.getDeviceIds();
    for (int deviceId : deviceIds) {
        InputDevice dev = InputDevice.getDevice(deviceId);
        int sources = dev.getSources();

        // Verify that the device has gamepad buttons, control sticks, or both.
        if (((sources & InputDevice.SOURCE_GAMEPAD) == InputDevice.SOURCE_GAMEPAD)
                || ((sources & InputDevice.SOURCE_JOYSTICK)
                == InputDevice.SOURCE_JOYSTICK)) {
            // This device is a game controller. Store its device ID.
            if (!gameControllerDeviceIds.contains(deviceId)) {
                gameControllerDeviceIds.add(deviceId);
            }
        }
    }
    return gameControllerDeviceIds;
}
```

Additionally, you might want to check for individual input capabilities supported by a connected game controller. This might be useful, for example, if you want your game to use only input from the set of physical controls it understands.

To detect if a specific key code or axis code is supported by a connected game controller, use these techniques:

- In Android 4.4 (API level 19) or higher, you can determine if a key code is supported on a connected game controller by calling `hasKeys(int)`.
- In Android 3.1 (API level 12) or higher, you can find all available axes supported on a connected game controller by first calling `getMotionRanges()`. Then, on each `InputDevice.MotionRange` object returned, call `getAxis()` to get its axis ID.

Process Gamepad Button Presses

Figure 1 shows how Android maps key codes and axis values to the physical controls on most game controllers.

Figure 1. Profile for a generic game controller.

The callouts in the figure refer to the following:

1. AXIS_HAT_X, AXIS_HAT_Y, DPAD_UP, DPAD_DOWN, DPAD_LEFT, DPAD_RIGHT
2. AXIS_X, AXIS_Y, BUTTON_THUMBL
3. AXIS_Z, AXIS_RZ, BUTTON_THUMBR
4. BUTTON_X
5. BUTTON_A
6. BUTTON_Y
7. BUTTON_B
8. BUTTON_R1
9. AXIS_RTRIGGER, AXIS_THROTTLE
10. AXIS_LTRIGGER, AXIS_BRAKE
11. BUTTON_L1

Common key codes generated by gamepad button presses include BUTTON_A, BUTTON_B, BUTTON_SELECT, and BUTTON_START. Some game controllers also trigger the DPAD_CENTER key code when the center of the D-pad crossbar is pressed. Your

game can inspect the key code by calling `getKeyCode()` or from key event callbacks such as `onKeyDown()`, and if it represents an event that is relevant to your game, process it as a game action. Table 1 lists the recommended game actions for the most common gamepad buttons.

Table 1. Recommended game actions for gamepad buttons.

Game Action	Button Key Code
Start game in main menu, or pause/unpause during game	`BUTTON_START`*
Display menu	`BUTTON_SELECT`* and `KEYCODE_MENU`*
Same as Android *Back* navigation behavior described in the Navigation design guide.	`KEYCODE_BACK`
Navigate back to a previous item in a menu	`BUTTON_B`
Confirm selection, or perform primary game action	`BUTTON_A` and `DPAD_CENTER`

** Your game should not rely on the presence of the Start, Select, or Menu buttons.*

Tip: Consider providing a configuration screen in your game to allow users to personalize their own game controller mappings for game actions.

The following snippet shows how you might override `onKeyDown()` to associate the `BUTTON_A` and `DPAD_CENTER` button presses with a game action.

```
public class GameView extends View {
    ...

    @Override
    public boolean onKeyDown(int keyCode, KeyEvent event) {
        boolean handled = false;
        if ((event.getSource() & InputDevice.SOURCE_GAMEPAD)
                == InputDevice.SOURCE_GAMEPAD) {
            if (event.getRepeatCount() == 0) {
                switch (keyCode) {
                    // Handle gamepad and D-pad button presses to
                    // navigate the ship
                    ...
```

```
                    default:
                         if (isFireKey(keyCode)) {
                             // Update the ship object to fire lasers
                             ...
                             handled = true;
                         }
                     break;
                }
            }
            if (handled) {
                return true;
            }
        }
        return super.onKeyDown(keyCode, event);
    }

    private static boolean isFireKey(int keyCode) {
        // Here we treat Button_A and DPAD_CENTER as the primary action
        // keys for the game.
        return keyCode == KeyEvent.KEYCODE_DPAD_CENTER
                || keyCode == KeyEvent.KEYCODE_BUTTON_A;
    }
}
```

Note: On Android 4.2 (API level 17) and lower, the system treats `BUTTON_A` as the Android *Back* key by default. If your app supports these Android versions, make sure to treat `BUTTON_A` as the primary game action. To determine the current Android SDK version on the device, refer to the `Build.VERSION.SDK_INT` value.

Process Directional Pad Input

The 4-way directional pad (D-pad) is a common physical control in many game controllers. Android reports D-pad UP and DOWN presses as `AXIS_HAT_Y` events with a range from -1.0 (up) to 1.0 (down), and D-pad LEFT or RIGHT presses as `AXIS_HAT_X` events with a range from -1.0 (left) to 1.0 (right).

Some controllers instead report D-pad presses with a key code. If your game cares about D-pad presses, you should treat the hat axis events and the D-pad key codes as the same input events, as recommended in table 2.

Table 2. Recommended default game actions for D-pad key codes and hat axis values.

Game Action	D-pad Key Code	Hat Axis Code
Move Up	KEYCODE_DPAD_UP	AXIS_HAT_Y (for values 0 to -1.0)
Move Down	KEYCODE_DPAD_DOWN	AXIS_HAT_Y (for values 0 to 1.0)
Move Left	KEYCODE_DPAD_LEFT	AXIS_HAT_X (for values 0 to -1.0)
Move Right	KEYCODE_DPAD_RIGHT	AXIS_HAT_X (for values 0 to 1.0)

The following code snippet shows a helper class that lets you check the hat axis and key code values from an input event to determine the D-pad direction.

```
public class Dpad {
    final static int UP       = 0;
    final static int LEFT     = 1;
    final static int RIGHT    = 2;
    final static int DOWN     = 3;
    final static int CENTER   = 4;

    int directionPressed = -1; // initialized to -1

    public int getDirectionPressed(InputEvent event) {
        if (!isDpadDevice(event)) {
           return -1;
        }

        // If the input event is a MotionEvent, check its hat axis values.
        if (event instanceof MotionEvent) {

            // Use the hat axis value to find the D-pad direction
            MotionEvent motionEvent = (MotionEvent) event;
            float xaxis = motionEvent.getAxisValue(MotionEvent.AXIS_HAT_X);
            float yaxis = motionEvent.getAxisValue(MotionEvent.AXIS_HAT_Y);

            // Check if the AXIS_HAT_X value is -1 or 1, and set the D-pad
            // LEFT and RIGHT direction accordingly.
            if (Float.compare(xaxis, -1.0f) == 0) {
                directionPressed =  Dpad.LEFT;
            } else if (Float.compare(xaxis, 1.0f) == 0) {
                directionPressed =  Dpad.RIGHT;
            }
            // Check if the AXIS_HAT_Y value is -1 or 1, and set the D-pad
            // UP and DOWN direction accordingly.
```

```
            else if (Float.compare(yaxis, -1.0f) == 0) {
                directionPressed =  Dpad.UP;
            } else if (Float.compare(yaxis, 1.0f) == 0) {
                directionPressed =  Dpad.DOWN;
            }
        }

        // If the input event is a KeyEvent, check its key code.
        else if (event instanceof KeyEvent) {

           // Use the key code to find the D-pad direction.
            KeyEvent keyEvent = (KeyEvent) event;
            if (keyEvent.getKeyCode() == KeyEvent.KEYCODE_DPAD_LEFT) {
                directionPressed = Dpad.LEFT;
            } else if (keyEvent.getKeyCode() == KeyEvent.KEYCODE_DPAD_RIGHT) {
                directionPressed = Dpad.RIGHT;
            } else if (keyEvent.getKeyCode() == KeyEvent.KEYCODE_DPAD_UP) {
                directionPressed = Dpad.UP;
            } else if (keyEvent.getKeyCode() == KeyEvent.KEYCODE_DPAD_DOWN) {
                directionPressed = Dpad.DOWN;
            } else if (keyEvent.getKeyCode() == KeyEvent.KEYCODE_DPAD_CENTER) {
                directionPressed = Dpad.CENTER;
            }
        }
        return directionPressed;
    }

    public static boolean isDpadDevice(InputEvent event) {
        // Check that input comes from a device with directional pads.
        if ((event.getSource() & InputDevice.SOURCE_DPAD)
             != InputDevice.SOURCE_DPAD) {
             return true;
         } else {
             return false;
         }
     }
}
```

You can use this helper class in your game wherever you want to process D-pad input (for example, in the `onGenericMotionEvent()` or `onKeyDown()` callbacks).

For example:

```
Dpad mDpad = new Dpad();
...
```

```
@Override
public boolean onGenericMotionEvent(MotionEvent event) {

    // Check if this event if from a D-pad and process accordingly.
    if (Dpad.isDpadDevice(event)) {

       int press = mDpad.getDirectionPressed(event);
       switch (press) {
            case LEFT:
                // Do something for LEFT direction press
                ...
                return true;
            case RIGHT:
                // Do something for RIGHT direction press
                ...
                return true;
            case UP:
                // Do something for UP direction press
                ...
                return true;
            ...
        }
    }

    // Check if this event is from a joystick movement and process accordingly.
    ...
}
```

Process Joystick Movements

When players move a joystick on their game controllers, Android reports a `MotionEvent` that contains the `ACTION_MOVE` action code and the updated positions of the joystick's axes. Your game can use the data provided by the `MotionEvent` to determine if a joystick movement it cares about happened.

Note that joystick motion events may batch multiple movement samples together within a single object. The `MotionEvent` object contains the current position for each joystick axis as well as multiple historical positions for each axis. When reporting motion events with action code `ACTION_MOVE` (such as joystick movements), Android batches up the axis values for efficiency. The historical values for an axis consists of the set of distinct values older than the current axis value, and more recent than values reported in any previous motion events. See the `MotionEvent` reference for

details.

You can use the historical information to more accurately render a game object's movement based on the joystick input. To retrieve the current and historical values, call `getAxisValue()` or `getHistoricalAxisValue()`. You can also find the number of historical points in the joystick event by calling `getHistorySize()`.

The following snippet shows how you might override the `onGenericMotionEvent()` callback to process joystick input. You should first process the historical values for an axis, then process its current position.

```
public class GameView extends View {

    @Override
    public boolean onGenericMotionEvent(MotionEvent event) {

        // Check that the event came from a game controller
        if ((event.getSource() & InputDevice.SOURCE_JOYSTICK) ==
                InputDevice.SOURCE_JOYSTICK &&
                event.getAction() == MotionEvent.ACTION_MOVE) {

            // Process all historical movement samples in the batch
            final int historySize = event.getHistorySize();

            // Process the movements starting from the
            // earliest historical position in the batch
            for (int i = 0; i < historySize; i++) {
                // Process the event at historical position i
                processJoystickInput(event, i);
            }

            // Process the current movement sample in the batch (position -1)
            processJoystickInput(event, -1);
            return true;
        }
        return super.onGenericMotionEvent(event);
    }
}
```

Before using joystick input, you need to determine if the joystick is centered, then calculate its axis movements accordingly. Joysticks typically have a *flat* area, that is, a range of values near the (0,0) coordinate at which the axis is considered to be centered. If the axis value reported by Android falls within the flat area, you should treat the controller to be at rest (that is, motionless along both axes).

The snippet below shows a helper method that calculates the movement along each axis. You invoke this helper in the `processJoystickInput()` method described further below.

```
private static float getCenteredAxis(MotionEvent event,
        InputDevice device, int axis, int historyPos) {
    final InputDevice.MotionRange range =
            device.getMotionRange(axis, event.getSource());

    // A joystick at rest does not always report an absolute position of
    // (0,0). Use the getFlat() method to determine the range of values
    // bounding the joystick axis center.
    if (range != null) {
        final float flat = range.getFlat();
        final float value =
                historyPos < 0 ? event.getAxisValue(axis):
                event.getHistoricalAxisValue(axis, historyPos);

        // Ignore axis values that are within the 'flat' region of the
        // joystick axis center.
        if (Math.abs(value) > flat) {
            return value;
        }
    }
    return 0;
}
```

Putting it all together, here is how you might process joystick movements in your game:

```
private void processJoystickInput(MotionEvent event,
        int historyPos) {

    InputDevice mInputDevice = event.getDevice();

    // Calculate the horizontal distance to move by
    // using the input value from one of these physical controls:
    // the left control stick, hat axis, or the right control stick.
    float x = getCenteredAxis(event, mInputDevice,
            MotionEvent.AXIS_X, historyPos);
    if (x == 0) {
        x = getCenteredAxis(event, mInputDevice,
                MotionEvent.AXIS_HAT_X, historyPos);
    }
    if (x == 0) {
        x = getCenteredAxis(event, mInputDevice,
```

```
                MotionEvent.AXIS_Z, historyPos);
    }

    // Calculate the vertical distance to move by
    // using the input value from one of these physical controls:
    // the left control stick, hat switch, or the right control stick.
    float y = getCenteredAxis(event, mInputDevice,
            MotionEvent.AXIS_Y, historyPos);
    if (y == 0) {
        y = getCenteredAxis(event, mInputDevice,
                MotionEvent.AXIS_HAT_Y, historyPos);
    }
    if (y == 0) {
        y = getCenteredAxis(event, mInputDevice,
                MotionEvent.AXIS_RZ, historyPos);
    }

    // Update the ship object based on the new x and y values
}
```

To support game controllers that have more sophisticated features beyond a single joystick, follow these best practices:

- **Handle dual controller sticks.** Many game controllers have both a left and right joystick. For the left stick, Android reports horizontal movements as `AXIS_X` events and vertical movements as `AXIS_Y` events. For the right stick, Android reports horizontal movements as `AXIS_Z` events and vertical movements as `AXIS_RZ` events. Make sure to handle both controller sticks in your code.
- **Handle shoulder trigger presses (but provide alternative input methods).** Some controllers have left and right shoulder triggers. If these triggers are present, Android reports a left trigger press as an `AXIS_LTRIGGER` event and a right trigger press as an `AXIS_RTRIGGER` event. On Android 4.3 (API level 18), a controller that produces a `AXIS_LTRIGGER` also reports an identical value for the `AXIS_BRAKE` axis. The same is true for `AXIS_RTRIGGER` and `AXIS_GAS`. Android reports all analog trigger presses with a normalized value from 0.0 (released) to 1.0 (fully pressed). Not all controllers have triggers, so consider allowing players to perform those game actions with other buttons.

Next Get started Next class

Supporting Controllers Across Android Versions

Previous Next Get started

This lesson teaches you to

1. Prepare to Abstract APIs for Game Controller Suppport
2. Add an Interface for Backward Compatibility
3. Implement the Interface on Android 4.1 and Higher
4. Implement the Interface on Android 2.3 up to Android 4.0
5. Use the Version-Specific Implementations

Try it out

Download the sample

ControllerSample.zip

If you are supporting game controllers in your game, it's your responsibility to make sure that your game responds to controllers consistently across devices running on different versions of Android. This lets your game reach a wider audience, and your players can enjoy a seamless gameplay experience with their controllers even when they switch or upgrade their Android devices.

This lesson demonstrates how to use APIs available in Android 4.1 and higher in a backward compatible way, enabling your game to support the following features on devices running Android 2.3 and higher:

- The game can detect if a new game controller is added, changed, or removed.
- The game can query the capabilities of a game controller.
- The game can recognize incoming motion events from a game

controller.

The examples in this lesson are based on the reference implementation provided by the sample `ControllerSample.zip` available for download above. This sample shows how to implement the `InputManagerCompat` interface to support different versions of Android. To compile the sample, you must use Android 4.1 (API level 16) or higher. Once compiled, the sample app runs on any device running Android 2.3 (API level 9) or higher as the build target.

Prepare to Abstract APIs for Game Controller Support

Suppose you want to be able to determine if a game controller's connection status has changed on devices running on Android 2.3 (API level 9). However, the APIs are only available in Android 4.1 (API level 16) and higher, so you need to provide an implementation that supports Android 4.1 and higher while providing a fallback mechanism that supports Android 2.3 up to Android 4.0.

To help you determine which features require such a fallback mechanism for older versions, table 1 lists the differences in game controller support between Android 2.3 (API level 9), 3.1 (API level 12), and 4.1 (API level 16).

Table 1. APIs for game controller support across different Android versions.

Controller Information	Controller API	API level 9	API level 12	API level 16
Device Identification	`getInputDeviceIds()`			•
	`getInputDevice()`			•
	`getVibrator()`			•
	`SOURCE_JOYSTICK`		•	•
	`SOURCE_GAMEPAD`		•	•
Connection Status	`onInputDeviceAdded()`			•
	`onInputDeviceChanged()`			•
	`onInputDeviceRemoved()`			•

Input Event Identification	D-pad press (KEYCODE_DPAD_UP, KEYCODE_DPAD_DOWN, KEYCODE_DPAD_LEFT, KEYCODE_DPAD_RIGHT, KEYCODE_DPAD_CENTER)	•	•	•
	Gamepad button press (BUTTON_A, BUTTON_B, BUTTON_THUMBL, BUTTON_THUMBR, BUTTON_SELECT, BUTTON_START, BUTTON_R1, BUTTON_L1, BUTTON_R2, BUTTON_L2)		•	•
	Joystick and hat switch movement (AXIS_X, AXIS_Y, AXIS_Z, AXIS_RZ, AXIS_HAT_X, AXIS_HAT_Y)		•	•
	Analog trigger press (AXIS_LTRIGGER, AXIS_RTRIGGER)		•	•

You can use abstraction to build version-aware game controller support that works across platforms. This approach involves the following steps:

1. Define an intermediary Java interface that abstracts the implementation of the game controller features required by your game.
2. Create a proxy implementation of your interface that uses APIs in Android 4.1 and higher.
3. Create a custom implementation of your interface that uses APIs available between Android 2.3 up to Android 4.0.
4. Create the logic for switching between these implementations at runtime, and begin using the interface in your game.

For an overview of how abstraction can be used to ensure that applications can work in a backward compatible way across different versions of Android, see Creating Backward-Compatible UIs.

Add an Interface for Backward Compatibility

To provide backward compatibility, you can create a custom interface then add version-specific implementations. One advantage of this approach is that it lets you mirror the public interfaces on Android 4.1 (API level 16) that support game controllers.

```
// The InputManagerCompat interface is a reference example.
// The full code is provided in the ControllerSample.zip sample.
public interface InputManagerCompat {
    ...
```

```
    public InputDevice getInputDevice(int id);
    public int[] getInputDeviceIds();

    public void registerInputDeviceListener(
            InputManagerCompat.InputDeviceListener listener,
            Handler handler);
    public void unregisterInputDeviceListener(
            InputManagerCompat.InputDeviceListener listener);

    public void onGenericMotionEvent(MotionEvent event);

    public void onPause();
    public void onResume();

    public interface InputDeviceListener {
        void onInputDeviceAdded(int deviceId);
        void onInputDeviceChanged(int deviceId);
        void onInputDeviceRemoved(int deviceId);
    }
    ...
}
```

The `InputManagerCompat` interface provides the following methods:

`getInputDevice()`
: Mirrors `getInputDevice()`. Obtains the `InputDevice` object that represents the capabilities of a game controller.

`getInputDeviceIds()`
: Mirrors `getInputDeviceIds()`. Returns an array of integers, each of which is an ID for a different input device. This is useful if you're building a game that supports multiple players and you want to detect how many controllers are connected.

`registerInputDeviceListener()`
: Mirrors `registerInputDeviceListener()`. Lets you register to be informed when a new device is added, changed, or removed.

`unregisterInputDeviceListener()`
: Mirrors `unregisterInputDeviceListener()`. Unregisters an input device listener.

`onGenericMotionEvent()`
: Mirrors `onGenericMotionEvent()`. Lets your game intercept and handle `MotionEvent` objects and axis values that represent events such as joystick movements and analog trigger presses.

`onPause()`
: Stops polling for game controller events when the main activity is paused, or when the game no longer has focus.

`onResume()`
: Starts polling for game controller events when the main activity is

resumed, or when the game is started and runs in the foreground.

`InputDeviceListener`
: Mirrors the `InputManager.InputDeviceListener` interface. Lets your game know when a game controller has been added, changed, or removed.

Next, create implementations for `InputManagerCompat` that work across different platform versions. If your game is running on Android 4.1 or higher and calls an `InputManagerCompat` method, the proxy implementation calls the equivalent method in `InputManager`. However, if your game is running on Android 2.3 up to Android 4.0, the custom implementation processes calls to `InputManagerCompat` methods by using only APIs introduced no later than Android 2.3. Regardless of which version-specific implementation is used at runtime, the implementation passes the call results back transparently to the game.

Figure 1. Class diagram of interface and version-specific implementations.

Implement the Interface on Android 4.1 and Higher

`InputManagerCompatV16` is an implementation of the `InputManagerCompat` interface that proxies method calls to an actual `InputManager` and

InputManager.InputDeviceListener. The InputManager is obtained from the system Context.

```
// The InputManagerCompatV16 class is a reference implementation.
// The full code is provided in the ControllerSample.zip sample.
public class InputManagerV16 implements InputManagerCompat {

    private final InputManager mInputManager;
    private final Map mListeners;

    public InputManagerV16(Context context) {
        mInputManager = (InputManager)
                context.getSystemService(Context.INPUT_SERVICE);
        mListeners = new HashMap();
    }

    @Override
    public InputDevice getInputDevice(int id) {
        return mInputManager.getInputDevice(id);
    }

    @Override
    public int[] getInputDeviceIds() {
        return mInputManager.getInputDeviceIds();
    }

    static class V16InputDeviceListener implements
            InputManager.InputDeviceListener {
        final InputManagerCompat.InputDeviceListener mIDL;

        public V16InputDeviceListener(InputDeviceListener idl) {
            mIDL = idl;
        }

        @Override
        public void onInputDeviceAdded(int deviceId) {
            mIDL.onInputDeviceAdded(deviceId);
        }

        // Do the same for device change and removal
        ...
    }

    @Override
    public void registerInputDeviceListener(InputDeviceListener
listener,
            Handler handler) {
        V16InputDeviceListener v16Listener = new
                V16InputDeviceListener(listener);
        mInputManager.registerInputDeviceListener(v16Listener, handler);
        mListeners.put(listener, v16Listener);
    }

    // Do the same for unregistering an input device listener
    ...

    @Override
    public void onGenericMotionEvent(MotionEvent event) {
        // unused in V16
    }

    @Override
```

```
    public void onPause() {
        // unused in V16
    }

    @Override
    public void onResume() {
        // unused in V16
    }

}
```

Implementing the Interface on Android 2.3 up to Android 4.0

The `InputManagerV9` implementation uses APIs introduced no later than Android 2.3. To create an implementation of `InputManagerCompat` that supports Android 2.3 up to Android 4.0, you can use the following objects:

- A `SparseArray` of device IDs to track the game controllers that are connected to the device.
- A `Handler` to process device events. When an app is started or resumed, the `Handler` receives a message to start polling for game controller disconnection. The `Handler` will start a loop to check each known connected game controller and see if a device ID is returned. A `null` return value indicates that the game controller is disconnected. The `Handler` stops polling when the app is paused.
- A `Map` of `InputManagerCompat.InputDeviceListener` objects. You will use the listeners to update the connection status of tracked game controllers.

```
// The InputManagerCompatV9 class is a reference implementation.
// The full code is provided in the ControllerSample.zip sample.
public class InputManagerV9 implements InputManagerCompat {
    private final SparseArray mDevices;
    private final Map mListeners;
    private final Handler mDefaultHandler;
    …

    public InputManagerV9() {
        mDevices = new SparseArray();
        mListeners = new HashMap();
        mDefaultHandler = new PollingMessageHandler(this);
    }
}
```

Implement a `PollingMessageHandler` object that extends `Handler`, and override the `handleMessage()` method. This method checks if an attached game controller has been disconnected and notifies registered listeners.

```
private static class PollingMessageHandler extends Handler {
    private final WeakReference mInputManager;

    PollingMessageHandler(InputManagerV9 im) {
        mInputManager = new WeakReference(im);
    }

    @Override
    public void handleMessage(Message msg) {
        super.handleMessage(msg);
        switch (msg.what) {
            case MESSAGE_TEST_FOR_DISCONNECT:
                InputManagerV9 imv = mInputManager.get();
                if (null != imv) {
                    long time = SystemClock.elapsedRealtime();
                    int size = imv.mDevices.size();
                    for (int i = 0; i < size; i++) {
                        long[] lastContact = imv.mDevices.valueAt(i);
                        if (null != lastContact) {
                            if (time - lastContact[0] >
CHECK_ELAPSED_TIME) {
                                // check to see if the device has been
                                // disconnected
                                int id = imv.mDevices.keyAt(i);
                                if (null == InputDevice.getDevice(id)) {
                                    // Notify the registered listeners
                                    // that the game controller is
disconnected
                                    ...
                                    imv.mDevices.remove(id);
                                } else {
                                    lastContact[0] = time;
                                }
                            }
                        }
                    }
                    sendEmptyMessageDelayed(MESSAGE_TEST_FOR_DISCONNECT,
                            CHECK_ELAPSED_TIME);
                }
                break;
        }
    }
}
```

To start and stop polling for game controller disconnection, override these methods:

```
private static final int MESSAGE_TEST_FOR_DISCONNECT = 101;
private static final long CHECK_ELAPSED_TIME = 3000L;

@Override
public void onPause() {
    mDefaultHandler.removeMessages(MESSAGE_TEST_FOR_DISCONNECT);
}

@Override
public void onResume() {
    mDefaultHandler.sendEmptyMessageDelayed(MESSAGE_TEST_FOR_DISCONNECT,
            CHECK_ELAPSED_TIME);
}
```

To detect that an input device has been added, override the `onGenericMotionEvent()` method. When the system reports a motion event, check if this event came from a device ID that is already tracked, or from a new device ID. If the device ID is new, notify registered listeners.

```
@Override
public void onGenericMotionEvent(MotionEvent event) {
    // detect new devices
    int id = event.getDeviceId();
    long[] timeArray = mDevices.get(id);
    if (null == timeArray) {
        // Notify the registered listeners that a game controller is
added
        ...
        timeArray = new long[1];
        mDevices.put(id, timeArray);
    }
    long time = SystemClock.elapsedRealtime();
    timeArray[0] = time;
}
```

Notification of listeners is implemented by using the `Handler` object to send a `DeviceEvent` `Runnable` object to the message queue. The `DeviceEvent` contains a reference to an `InputManagerCompat.InputDeviceListener`. When the `DeviceEvent` runs, the appropriate callback method of the listener is called to signal if the game controller was added, changed, or removed.

```
@Override
public void registerInputDeviceListener(InputDeviceListener listener,
        Handler handler) {
    mListeners.remove(listener);
    if (handler == null) {
        handler = mDefaultHandler;
    }
    mListeners.put(listener, handler);
}

@Override
public void unregisterInputDeviceListener(InputDeviceListener listener)
{
    mListeners.remove(listener);
}

private void notifyListeners(int why, int deviceId) {
    // the state of some device has changed
    if (!mListeners.isEmpty()) {
        for (InputDeviceListener listener : mListeners.keySet()) {
            Handler handler = mListeners.get(listener);
            DeviceEvent odc = DeviceEvent.getDeviceEvent(why, deviceId,
                    listener);
            handler.post(odc);
        }
    }
}

private static class DeviceEvent implements Runnable {
    private int mMessageType;
```

```
    private int mId;
    private InputDeviceListener mListener;
    private static Queue sObjectQueue =
            new ArrayDeque();
    ...

    static DeviceEvent getDeviceEvent(int messageType, int id,
            InputDeviceListener listener) {
        DeviceEvent curChanged = sObjectQueue.poll();
        if (null == curChanged) {
            curChanged = new DeviceEvent();
        }
        curChanged.mMessageType = messageType;
        curChanged.mId = id;
        curChanged.mListener = listener;
        return curChanged;
    }

    @Override
    public void run() {
        switch (mMessageType) {
            case ON_DEVICE_ADDED:
                mListener.onInputDeviceAdded(mId);
                break;
            case ON_DEVICE_CHANGED:
                mListener.onInputDeviceChanged(mId);
                break;
            case ON_DEVICE_REMOVED:
                mListener.onInputDeviceRemoved(mId);
                break;
            default:
                // Handle unknown message type
                ...
                break;
        }
        // Put this runnable back in the queue
        sObjectQueue.offer(this);
    }
}
```

You now have two implementations of `InputManagerCompat`: one that works on devices running Android 4.1 and higher, and another that works on devices running Android 2.3 up to Android 4.0.

Use the Version-Specific Implementation

The version-specific switching logic is implemented in a class that acts as a factory.

```
public static class Factory {
    public static InputManagerCompat getInputManager(Context context) {
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN) {
            return new InputManagerV16(context);
        } else {
            return new InputManagerV9();
        }
    }
```

```
}
```

Now you can simply instantiate an `InputManagerCompat` object and register an `InputManagerCompat.InputDeviceListener` in your main `View`. Because of the version-switching logic you set up, your game automatically uses the implementation that's appropriate for the version of Android the device is running.

```
public class GameView extends View implements InputDeviceListener {
    private InputManagerCompat mInputManager;
    ...

    public GameView(Context context, AttributeSet attrs) {
        mInputManager =

InputManagerCompat.Factory.getInputManager(this.getContext());
        mInputManager.registerInputDeviceListener(this, null);
        ...
    }
}
```

Next, override the `onGenericMotionEvent()` method in your main view, as described in Handle a MotionEvent from a Game Controller. Your game should now be able to process game controller events consistently on devices running Android 2.3 (API level 9) and higher.

```
@Override
public boolean onGenericMotionEvent(MotionEvent event) {
    mInputManager.onGenericMotionEvent(event);

    // Handle analog input from the controller as normal
    ...
    return super.onGenericMotionEvent(event);
}
```

You can find a complete implementation of this compatibility code in the `GameView` class provided in the sample `ControllerSample.zip` available for download above.

Next Get started Next class

Supporting Multiple Game Controllers

Previous Next Get started

This lesson teaches you to

1. Map Players to Controller Device IDs
2. Process Multiple Controller Input

Try it out

Download the sample

ControllerSample.zip

While most games are designed to support a single user per Android device, it's also possible to support multiple users with game controllers that are connected simultaneously on the same Android device.

This lesson covers some basic techniques for handling input in your single device multiplayer game from multiple connected controllers. This includes maintaining a mapping between player avatars and each controller device and processing controller input events appropriately.

Map Players to Controller Device IDs

When a game controller is connected to an Android device, the system assigns it an integer device ID. You can obtain the device IDs for connected game controllers by calling `InputDevice.getDeviceIds()`, as shown in Verify a Game Controller is Connected. You can then associate each device ID with a player in your game, and process game actions for

each player separately.

Note: On devices running Android 4.1 (API level 16) and higher, you can obtain an input device's descriptor using `getDescriptor()`, which returns a unique persistent string value for the input device. Unlike a device ID, the descriptor value won't change even if the input device is disconnected, reconnected, or reconfigured.

The code snippet below shows how to use a `SparseArray` to associate a player's avatar with a specific controller. In this example, the `mShips` variable stores a collection of `Ship` objects. A new player avatar is created in-game when a new controller is attached by a user, and removed when its associated controller is removed.

The `onInputDeviceAdded()` and `onInputDeviceRemoved()` callback methods are part of the abstraction layer introduced in Supporting Controllers Across Android Versions. By implementing these listener callbacks, your game can identify the game controller's device ID when a controller is added or removed. This detection is compatible with Android 2.3 (API level 9) and higher.

```
private final SparseArray<Ship> mShips = new SparseArray<Ship>();

@Override
public void onInputDeviceAdded(int deviceId) {
    getShipForID(deviceId);
}

@Override
public void onInputDeviceRemoved(int deviceId) {
    removeShipForID(deviceId);
}

private Ship getShipForID(int shipID) {
    Ship currentShip = mShips.get(shipID);
    if ( null == currentShip ) {
        currentShip = new Ship();
        mShips.append(shipID, currentShip);
    }
    return currentShip;
}

private void removeShipForID(int shipID) {
    mShips.remove(shipID);
}
```

Process Multiple Controller Input

Your game should execute the following loop to process input from multiple controllers:

1. Detect whether an input event occurred.
2. Identify the input source and its device ID.
3. Based on the action indicated by the input event key code or axis value, update the player avatar associated with that device ID.
4. Render and update the user interface.

KeyEvent and MotionEvent input events have device IDs associated with them. Your game can take advantage of this to determine which controller the input event came from, and update the player avatar associated with that controller.

The following code snippet shows how you might get a player avatar reference corresponding to a game controller device ID, and update the game based on the user's button press on that controller.

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
    if ((event.getSource() & InputDevice.SOURCE_GAMEPAD)
                == InputDevice.SOURCE_GAMEPAD) {
        int deviceId = event.getDeviceId();
        if (deviceId != -1) {
            Ship currentShip = getShipForId(deviceId);
            // Based on which key was pressed, update the player
avatar
            // (e.g. set the ship headings or fire lasers)
            ...
            return true;
        }
    }
    return super.onKeyDown(keyCode, event);
}
```

Note: As a best practice, when a user's game controller disconnects, you should pause the game and ask if the user wants to reconnect.

Next Get started Next class

Best Practices for Background Jobs

These classes show you how to run jobs in the background to boost your application's performance and minimize its drain on the battery.

Running in a Background Service

Previous Next Get started

Dependencies and prerequisites

- Android 1.6 (API Level 4) or higher

You should also read

- Extending the IntentService Class
- Intents and Intent Filters

Try it out

Download the sample

ThreadSample.zip

Unless you specify otherwise, most of the operations you do in an app run in the foreground on a special thread called the UI thread. This can cause problems, because long-running operations will interfere with the responsiveness of your user interface. This annoys your users, and can even cause system errors. To avoid this, the Android framework offers several classes that help you off-load operations onto a separate thread running in the background. The most useful of these is `IntentService`.

This class describes how to implement an `IntentService`, send it work requests, and report its results to other components.

Lessons

Creating a Background Service

Learn how to create an `IntentService`.

Sending Work Requests to the Background Service

Learn how to send work requests to an `IntentService`.

Reporting Work Status

Learn how to use an `Intent` and a `LocalBroadcastManager` to communicate the status of a work request from an `IntentService` to the `Activity` that sent the request.

Next Get started Next class

Creating a Background Service

Previous Next Get started

This lesson teaches you to

1. Create an IntentService
2. Define the IntentService in the Manifest

You should also read

- Extending the IntentService Class
- Intents and Intent Filters

Try it out

Download the sample

ThreadSample.zip

The `IntentService` class provides a straightforward structure for running an operation on a single background thread. This allows it to handle long-running operations without affecting your user interface's responsiveness. Also, an `IntentService` isn't affected by most user interface lifecycle events, so it continues to run in circumstances that would shut down an `AsyncTask`

An `IntentService` has a few limitations:

- It can't interact directly with your user interface. To put its results in the UI, you have to send them to an `Activity`.
- Work requests run sequentially. If an operation is running in an `IntentService`, and you send it another request, the request waits

until the first operation is finished.
- An operation running on an `IntentService` can't be interrupted.

However, in most cases an `IntentService` is the preferred way to perform simple background operations.

This lesson shows you how to create your own subclass of `IntentService`. The lesson also shows you how to create the required callback method `onHandleIntent()`. Finally, the lesson describes shows you how to define the `IntentService` in your manifest file.

Create an IntentService

To create an `IntentService` component for your app, define a class that extends `IntentService`, and within it, define a method that overrides `onHandleIntent()`. For example:

```
public class RSSPullService extends IntentService {
    @Override
    protected void onHandleIntent(Intent workIntent) {
        // Gets data from the incoming Intent
        String dataString = workIntent.getDataString();
        ...
        // Do work here, based on the contents of dataString
        ...
    }
}
```

Notice that the other callbacks of a regular `Service` component, such as `onStartCommand()` are automatically invoked by `IntentService`. In an `IntentService`, you should avoid overriding these callbacks.

Define the IntentService in the Manifest

An `IntentService` also needs an entry in your application manifest. Provide this entry as a `<service>` element that's a child of the `<application>` element:

```
    <application
        android:icon="@drawable/icon"
        android:label="@string/app_name">
        ...
        <!--
```

```
            Because android:exported is set to "false",
            the service is only available to this app.
        -->
        <service
            android:name=".RSSPullService"
            android:exported="false"/>
        ...
    <application/>
```

The attribute `android:name` specifies the class name of the `IntentService`.

Notice that the `<service>` element doesn't contain an intent filter. The `Activity` that sends work requests to the service uses an explicit `Intent`, so no filter is needed. This also means that only components in the same app or other applications with the same user ID can access the service.

Now that you have the basic `IntentService` class, you can send work requests to it with `Intent` objects. The procedure for constructing these objects and sending them to your `IntentService` is described in the next lesson.

Next Get started Next class

Sending Work Requests to the Background Service

Previous Next Get started

This lesson teaches you to

1. Create and Send a Work Request to an IntentService

You should also read

- Intents and Intent Filters

Try it out

Download the sample

ThreadSample.zip

The previous lesson showed you how to create an `IntentService` class. This lesson shows you how to trigger the `IntentService` to run an operation by sending it an `Intent`. This `Intent` can optionally contain data for the `IntentService` to process. You can send an `Intent` to an `IntentService` from any point in an `Activity` or `Fragment`

Create and Send a Work Request to an IntentService

To create a work request and send it to an `IntentService`, create an explicit `Intent`, add work request data to it, and send it to `IntentService` by calling `startService()`.

The next snippets demonstrate this:

1. Create a new, explicit `Intent` for the `IntentService` called `RSSPullService`.

   ```
   /*
    * Creates a new Intent to start the RSSPullService
    * IntentService. Passes a URI in the
    * Intent's "data" field.
    */
   mServiceIntent = new Intent(getActivity(),
   RSSPullService.class);
   mServiceIntent.setData(Uri.parse(dataUrl));
   ```

2. Call `startService()`

   ```
   // Starts the IntentService
   getActivity().startService(mServiceIntent);
   ```

Notice that you can send the work request from anywhere in an Activity or Fragment. For example, if you need to get user input first, you can send the request from a callback that responds to a button click or similar gesture.

Once you call `startService()`, the `IntentService` does the work defined in its `onHandleIntent()` method, and then stops itself.

The next step is to report the results of the work request back to the originating Activity or Fragment. The next lesson shows you how to do this with a `BroadcastReceiver`.

Next Get started Next class

Reporting Work Status

Previous Next Get started

This lesson teaches you to

1. Report Status From an IntentService
2. Receive Status Broadcasts from an IntentService

You should also read

- Intents and Intent Filters
- The section **Broadcast receivers** in the Application Components API guide.

Try it out

Download the sample

ThreadSample.zip

This lesson shows you how to report the status of a work request run in a background service to the component that sent the request. This allows you, for example, to report the status of the request in an `Activity` object's UI. The recommended way to send and receive status is to use a `LocalBroadcastManager`, which limits broadcast `Intent` objects to components in your own app.

Report Status From an IntentService

To send the status of a work request in an `IntentService` to other components, first create an `Intent` that contains the status in its

extended data. As an option, you can add an action and data URI to this Intent.

Next, send the Intent by calling LocalBroadcastManager.sendBroadcast(). This sends the Intent to any component in your application that has registered to receive it. To get an instance of LocalBroadcastManager, call getInstance().

For example:

```
public final class Constants {
    ...
    // Defines a custom Intent action
    public static final String BROADCAST_ACTION =
        "com.example.android.threadsample.BROADCAST";
    ...
    // Defines the key for the status "extra" in an Intent
    public static final String EXTENDED_DATA_STATUS =
        "com.example.android.threadsample.STATUS";
    ...
}
public class RSSPullService extends IntentService {
...
    /*
     * Creates a new Intent containing a Uri object
     * BROADCAST_ACTION is a custom Intent action
     */
    Intent localIntent =
            new Intent(Constants.BROADCAST_ACTION)
            // Puts the status into the Intent
            .putExtra(Constants.EXTENDED_DATA_STATUS, status);
    // Broadcasts the Intent to receivers in this app.

LocalBroadcastManager.getInstance(this).sendBroadcast(localIntent);
...
}
```

The next step is to handle the incoming broadcast Intent objects in the component that sent the original work request.

Receive Status Broadcasts from an IntentService

To receive broadcast Intent objects, use a subclass of BroadcastReceiver. In the subclass, implement the BroadcastReceiver.onReceive() callback method, which LocalBroadcastManager invokes when it receives an Intent. LocalBroadcastManager passes the incoming Intent to

BroadcastReceiver.onReceive().

For example:

```
// Broadcast receiver for receiving status updates from the
IntentService
private class ResponseReceiver extends BroadcastReceiver
{
    // Prevents instantiation
    private DownloadStateReceiver() {
    }
    // Called when the BroadcastReceiver gets an Intent it's
registered to receive
    @
    public void onReceive(Context context, Intent intent) {
...
        /*
         * Handle Intents here.
         */
...
    }
}
```

Once you've defined the BroadcastReceiver, you can define filters for it that match specific actions, categories, and data. To do this, create an IntentFilter. This first snippet shows how to define the filter:

```
// Class that displays photos
public class DisplayActivity extends FragmentActivity {
    ...
    public void onCreate(Bundle stateBundle) {
        ...
        super.onCreate(stateBundle);
        ...
        // The filter's action is BROADCAST_ACTION
        IntentFilter mStatusIntentFilter = new IntentFilter(
                Constants.BROADCAST_ACTION);

        // Adds a data filter for the HTTP scheme
        mStatusIntentFilter.addDataScheme("http");
        ...
```

To register the BroadcastReceiver and the IntentFilter with the system, get an instance of LocalBroadcastManager and call its registerReceiver() method. This next snippet shows how to register the BroadcastReceiver and its IntentFilter:

```
        // Instantiates a new DownloadStateReceiver
        DownloadStateReceiver mDownloadStateReceiver =
                new DownloadStateReceiver();
        // Registers the DownloadStateReceiver and its intent
filters
        LocalBroadcastManager.getInstance(this).registerReceiver(
```

```
                mDownloadStateReceiver,
                mStatusIntentFilter);
        ...
```

A single BroadcastReceiver can handle more than one type of broadcast Intent object, each with its own action. This feature allows you to run different code for each action, without having to define a separate BroadcastReceiver for each action. To define another IntentFilter for the same BroadcastReceiver, create the IntentFilter and repeat the call to registerReceiver(). For example:

```
        /*
         * Instantiates a new action filter.
         * No data filter is needed.
         */
        statusIntentFilter = new
IntentFilter(Constants.ACTION_ZOOM_IMAGE);
        ...
        // Registers the receiver with the new filter

LocalBroadcastManager.getInstance(getActivity()).registerReceiver(
                mDownloadStateReceiver,
                mIntentFilter);
```

Sending an broadcast Intent doesn't start or resume an Activity. The BroadcastReceiver for an Activity receives and processes Intent objects even when your app is in the background, but doesn't force your app to the foreground. If you want to notify the user about an event that happened in the background while your app was not visible, use a Notification. *Never* start an Activity in response to an incoming broadcast Intent.

Next Get started Next class

Loading Data in the Background

Previous Next Get started

Dependencies and prerequisites

- Android 1.6 or later

You should also read

- Loaders
- Using Databases
- Content Provider Basics

Try it out

Download the sample

ThreadSample.zip

Querying a `ContentProvider` for data you want to display takes time. If you run the query directly from an `Activity`, it may get blocked and cause the system to issue an "Application Not Responding" message. Even if it doesn't, users will see an annoying delay in the UI. To avoid these problems, you should initiate a query on a separate thread, wait for it to finish, and then display the results.

You can do this in a straightforward way by using an object that runs a query asynchronously in the background and reconnects to your `Activity` when it's finished. This object is a `CursorLoader`. Besides doing the initial background query, a `CursorLoader` automatically re-runs the query when data associated with the query changes.

This class describes how to use a `CursorLoader` to run a background query. Examples in this class use the v4 Support Library versions of classes, which support platforms starting with Android 1.6.

Lessons

Running a Query with a CursorLoader

Learn how to run a query in the background, using a `CursorLoader`.

Handling the Results

Learn how to handle the `Cursor` returned from the query, and how to remove references to the current `Cursor` when the loader framework re-sets the `CursorLoader`.

Next Get started Next class

Running a Query with a CursorLoader

Previous Next Get started

This lesson teaches you to

1. Define an Activity That Uses CursorLoader
2. Initialize the Query
3. Start the Query

Try it out

Download the sample

ThreadSample.zip

A `CursorLoader` runs an asynchronous query in the background against a `ContentProvider`, and returns the results to the `Activity` or `FragmentActivity` from which it was called. This allows the `Activity` or `FragmentActivity` to continue to interact with the user while the query is ongoing.

Define an Activity That Uses CursorLoader

To use a `CursorLoader` with an `Activity` or `FragmentActivity`, use the `LoaderCallbacks<Cursor>` interface. A `CursorLoader` invokes callbacks defined in this interface to communicate with the class; this lesson and the next one describe each callback in detail.

For example, this is how you should define a `FragmentActivity` that uses the support library version of `CursorLoader`. By extending

FragmentActivity, you get support for CursorLoader as well as Fragment:

```
public class PhotoThumbnailFragment extends FragmentActivity
implements
        LoaderManager.LoaderCallbacks<Cursor> {
...
}
```

Initialize the Query

To initialize a query, call LoaderManager.initLoader(). This initializes the background framework. You can do this after the user has entered data that's used in the query, or, if you don't need any user data, you can do it in onCreate() or onCreateView(). For example:

```
    // Identifies a particular Loader being used in this component
    private static final int URL_LOADER = 0;
    ...
    /* When the system is ready for the Fragment to appear, this
displays
     * the Fragment's View
     */
    public View onCreateView(
            LayoutInflater inflater,
            ViewGroup viewGroup,
            Bundle bundle) {
        ...
        /*
         * Initializes the CursorLoader. The URL_LOADER value is
eventually passed
         * to onCreateLoader().
         */
        getLoaderManager().initLoader(URL_LOADER, null, this);
        ...
    }
```

Note: The method getLoaderManager() is only available in the Fragment class. To get a LoaderManager in a FragmentActivity, call getSupportLoaderManager().

Start the Query

As soon as the background framework is initialized, it calls your implementation of onCreateLoader(). To start the query, return a CursorLoader from this method. You can instantiate an empty CursorLoader and then use its methods to define your query, or you can

instantiate the object and define the query at the same time:

```
/*
* Callback that's invoked when the system has initialized the Loader
and
* is ready to start the query. This usually happens when
initLoader() is
* called. The loaderID argument contains the ID value passed to the
* initLoader() call.
*/
@Override
public Loader<Cursor> onCreateLoader(int loaderID, Bundle bundle)
{
    /*
     * Takes action based on the ID of the Loader that's being
created
     */
    switch (loaderID) {
        case URL_LOADER:
            // Returns a new CursorLoader
            return new CursorLoader(
                        getActivity(),   // Parent activity context
                        mDataUrl,        // Table to query
                        mProjection,     // Projection to return
                        null,            // No selection clause
                        null,            // No selection arguments
                        null             // Default sort order
        );
        default:
            // An invalid id was passed in
            return null;
    }
}
```

Once the background framework has the object, it starts the query in the background. When the query is done, the background framework calls `onLoadFinished()`, which is described in the next lesson.

Next Get started Next class

Handling the Results

Previous Next Get started

This lesson teaches you to

1. Handle Query Results
2. Delete Old Cursor References

Try it out

Download the sample

ThreadSample.zip

As shown in the previous lesson, you should begin loading your data with a `CursorLoader` in your implementation of `onCreateLoader()`. The loader then provides the query results to your `Activity` or `FragmentActivity` in your implementation of `LoaderCallbacks.onLoadFinished()`. One of the incoming arguments to this method is a `Cursor` containing the query results. You can use this object to update your data display or do further processing.

Besides `onCreateLoader()` and `onLoadFinished()`, you also have to implement `onLoaderReset()`. This method is invoked when `CursorLoader` detects that data associated with the `Cursor` has changed. When the data changes, the framework also re-runs the current query.

Handle Query Results

To display `Cursor` data returned by `CursorLoader`, use a `View` class that implements `AdapterView` and provide the view with an adapter that implements `CursorAdapter`. The system then automatically moves data

from the Cursor to the view.

You can set up the linkage between the view and adapter before you have any data to display, and then move a Cursor into the adapter in the onLoadFinished() method. As soon as you move the Cursor into the adapter, the system automatically updates the view. This also happens if you change the contents of the Cursor.

For example:

```
public String[] mFromColumns = {
    DataProviderContract.IMAGE_PICTURENAME_COLUMN
};
public int[] mToFields = {
    R.id.PictureName
};
// Gets a handle to a List View
ListView mListView = (ListView) findViewById(R.id.dataList);
/*
 * Defines a SimpleCursorAdapter for the ListView
 *
 */
SimpleCursorAdapter mAdapter =
    new SimpleCursorAdapter(
            this,                   // Current context
            R.layout.list_item,     // Layout for a single row
            null,                   // No Cursor yet
            mFromColumns,           // Cursor columns to use
            mToFields,              // Layout fields to use
            0                       // No flags
    );
// Sets the adapter for the view
mListView.setAdapter(mAdapter);
...
/*
 * Defines the callback that CursorLoader calls
 * when it's finished its query
 */
@Override
public void onLoadFinished(Loader<Cursor> loader, Cursor cursor) {
    ...
    /*
     * Moves the query results into the adapter, causing the
     * ListView fronting this adapter to re-display
     */
    mAdapter.changeCursor(cursor);
}
```

Delete Old Cursor References

The CursorLoader is reset whenever its Cursor becomes invalid. This usually occurs because the data associated with the Cursor has changed.

Before re-running the query, the framework calls your implementation of `onLoaderReset()`. In this callback, you should delete all references to the current `Cursor` in order to prevent memory leaks. Once `onLoaderReset()` finishes, `CursorLoader` re-runs its query.

For example:

```
/*
 * Invoked when the CursorLoader is being reset. For example, this
is
 * called if the data in the provider changes and the Cursor becomes
stale.
 */
@Override
public void onLoaderReset(Loader<Cursor> loader) {

    /*
     * Clears out the adapter's reference to the Cursor.
     * This prevents memory leaks.
     */
    mAdapter.changeCursor(null);
}
```

Next Get started Next class

Managing Device Awake State

Previous Next Get started

Dependencies and prerequisites

- Android 1.6 (API Level 4) or higher

Try it out

Download the sample

Scheduler.zip

When an Android device is left idle, it will first dim, then turn off the screen, and ultimately turn off the CPU. This prevents the device's battery from quickly getting drained. Yet there are times when your application might require a different behavior:

- Apps such as games or movie apps may need to keep the screen turned on.
- Other applications may not need the screen to remain on, but they may require the CPU to keep running until a critical operation finishes.

This class describes how to keep a device awake when necessary without draining its battery.

Lessons

Keeping the Device Awake

Learn how to keep the screen or CPU awake as needed, while

minimizing the impact on battery life.

Scheduling Repeating Alarms

Learn how to use repeating alarms to schedule operations that take place outside of the lifetime of the application, even if the application is not running and/or the device is asleep.

Next Get started Next class

Keeping the Device Awake

Previous Next Get started

This lesson teaches you to

1. Keep the Screen On
2. Keep the CPU On

Try it out

Download the sample

Scheduler.zip

To avoid draining the battery, an Android device that is left idle quickly falls asleep. However, there are times when an application needs to wake up the screen or the CPU and keep it awake to complete some work.

The approach you take depends on the needs of your app. However, a general rule of thumb is that you should use the most lightweight approach possible for your app, to minimize your app's impact on system resources. The following sections describe how to handle the cases where the device's default sleep behavior is incompatible with the requirements of your app.

Keep the Screen On

Certain apps need to keep the screen turned on, such as games or movie apps. The best way to do this is to use the `FLAG_KEEP_SCREEN_ON` in your activity (and only in an activity, never in a service or other app component). For example:

```
public class MainActivity extends Activity {
```

```
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
  }
```

The advantage of this approach is that unlike wake locks (discussed in Keep the CPU On), it doesn't require special permission, and the platform correctly manages the user moving between applications, without your app needing to worry about releasing unused resources.

Another way to implement this is in your application's layout XML file, by using the `android:keepScreenOn` attribute:

```
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:keepScreenOn="true">
    ...
</RelativeLayout>
```

Using `android:keepScreenOn="true"` is equivalent to using `FLAG_KEEP_SCREEN_ON`. You can use whichever approach is best for your app. The advantage of setting the flag programmatically in your activity is that it gives you the option of programmatically clearing the flag later and thereby allowing the screen to turn off.

Note: You don't need to clear the `FLAG_KEEP_SCREEN_ON` flag unless you no longer want the screen to stay on in your running application (for example, if you want the screen to time out after a certain period of inactivity). The window manager takes care of ensuring that the right things happen when the app goes into the background or returns to the foreground. But if you want to explicitly clear the flag and thereby allow the screen to turn off again, use `clearFlags()`:

```
getWindow().clearFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON)
```

.

Keep the CPU On

Alternatives to using wake locks

- If your app is performing long-running HTTP downloads, consider using `DownloadManager`.
- If your app is synchronizing data from an external server, consider creating a sync adapter.
- If your app relies on background services, consider using repeating alarms or Google Cloud Messaging to trigger these services at specific intervals.

If you need to keep the CPU running in order to complete some work before the device goes to sleep, you can use a `PowerManager` system service feature called wake locks. Wake locks allow your application to control the power state of the host device.

Creating and holding wake locks can have a dramatic impact on the host device's battery life. Thus you should use wake locks only when strictly necessary and hold them for as short a time as possible. For example, you should never need to use a wake lock in an activity. As described above, if you want to keep the screen on in your activity, use `FLAG_KEEP_SCREEN_ON`.

One legitimate case for using a wake lock might be a background service that needs to grab a wake lock to keep the CPU running to do work while the screen is off. Again, though, this practice should be minimized because of its impact on battery life.

To use a wake lock, the first step is to add the `WAKE_LOCK` permission to your application's manifest file:

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

If your app includes a broadcast receiver that uses a service to do some work, you can manage your wake lock through a `WakefulBroadcastReceiver`, as described in Using a WakefulBroadcastReceiver. This is the preferred approach. If your app doesn't follow that pattern, here is how you set a wake lock directly:

```
PowerManager powerManager = (PowerManager)
getSystemService(POWER_SERVICE);
Wakelock wakeLock =
```

```
powerManager.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK,
        "MyWakelockTag");
wakeLock.acquire();
```

To release the wake lock, call `wakelock.release()`. This releases your claim to the CPU. It's important to release a wake lock as soon as your app is finished using it to avoid draining the battery.

Using WakefulBroadcastReceiver

Using a broadcast receiver in conjunction with a service lets you manage the life cycle of a background task.

A `WakefulBroadcastReceiver` is a special type of broadcast receiver that takes care of creating and managing a `PARTIAL_WAKE_LOCK` for your app. A `WakefulBroadcastReceiver` passes off the work to a `Service` (typically an `IntentService`), while ensuring that the device does not go back to sleep in the transition. If you don't hold a wake lock while transitioning the work to a service, you are effectively allowing the device to go back to sleep before the work completes. The net result is that the app might not finish doing the work until some arbitrary point in the future, which is not what you want.

The first step in using a `WakefulBroadcastReceiver` is to add it to your manifest, as with any other broadcast receiver:

```
<receiver android:name=".MyWakefulReceiver"></receiver>
```

The following code starts `MyIntentService` with the method `startWakefulService()`. This method is comparable to `startService()`, except that the `WakefulBroadcastReceiver` is holding a wake lock when the service starts. The intent that is passed with `startWakefulService()` holds an extra identifying the wake lock:

```
public class MyWakefulReceiver extends WakefulBroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {

        // Start the service, keeping the device awake while the
service is
        // launching. This is the Intent to deliver to the service.
        Intent service = new Intent(context, MyIntentService.class);
        startWakefulService(context, service);
```

```
    }
}
```

When the service is finished, it calls `MyWakefulReceiver.completeWakefulIntent()` to release the wake lock. The `completeWakefulIntent()` method has as its parameter the same intent that was passed in from the `WakefulBroadcastReceiver`:

```
public class MyIntentService extends IntentService {
    public static final int NOTIFICATION_ID = 1;
    private NotificationManager mNotificationManager;
    NotificationCompat.Builder builder;
    public MyIntentService() {
        super("MyIntentService");
    }
    @Override
    protected void onHandleIntent(Intent intent) {
        Bundle extras = intent.getExtras();
        // Do the work that requires your app to keep the CPU
running.
        // ...
        // Release the wake lock provided by the
WakefulBroadcastReceiver.
        MyWakefulReceiver.completeWakefulIntent(intent);
    }
}
```

Next Get started Next class

Scheduling Repeating Alarms

Previous Next Get started

This lesson teaches you to

1. Understand the Trade-offs
2. Set a Repeating Alarm
3. Cancel an Alarm
4. Start an Alarm When the Device Boots

Try it out

Download the sample

Scheduler.zip

Video

The App Clinic: Cricket

Video

DevBytes: Efficient Data Transfers

Alarms (based on the `AlarmManager` class) give you a way to perform time-based operations outside the lifetime of your application. For example, you could use an alarm to initiate a long-running operation, such as starting a service once a day to download a weather forecast.

Alarms have these characteristics:

- They let you fire Intents at set times and/or intervals.
- You can use them in conjunction with broadcast receivers to start

services and perform other operations.
- They operate outside of your application, so you can use them to trigger events or actions even when your app is not running, and even if the device itself is asleep.
- They help you to minimize your app's resource requirements. You can schedule operations without relying on timers or continuously running background services.

Note: For timing operations that are guaranteed to occur *during* the lifetime of your application, instead consider using the `Handler` class in conjunction with `Timer` and `Thread`. This approach gives Android better control over system resources.

Understand the Trade-offs

A repeating alarm is a relatively simple mechanism with limited flexibility. It may not be the best choice for your app, particularly if you need to trigger network operations. A poorly designed alarm can cause battery drain and put a significant load on servers.

A common scenario for triggering an operation outside the lifetime of your app is syncing data with a server. This is a case where you might be tempted to use a repeating alarm. But if you own the server that is hosting your app's data, using Google Cloud Messaging (GCM) in conjunction with sync adapter is a better solution than `AlarmManager`. A sync adapter gives you all the same scheduling options as `AlarmManager`, but it offers you significantly more flexibility. For example, a sync could be based on a "new data" message from the server/device (see Running a Sync Adapter for details), the user's activity (or inactivity), the time of day, and so on. See the linked videos at the top of this page for a detailed discussion of when and how to use GCM and sync adapter.

Best practices

Every choice you make in designing your repeating alarm can have consequences in how your app uses (or abuses) system resources. For example, imagine a popular app that syncs with a server. If the sync

operation is based on clock time and every instance of the app syncs at 11:00 p.m., the load on the server could result in high latency or even "denial of service." Follow these best practices in using alarms:

- Add randomness (jitter) to any network requests that trigger as a result of a repeating alarm:
  - Do any local work when the alarm triggers. "Local work" means anything that doesn't hit a server or require the data from the server.
  - At the same time, schedule the alarm that contains the network requests to fire at some random period of time.
- Keep your alarm frequency to a minimum.
- Don't wake up the device unnecessarily (this behavior is determined by the alarm type, as described in Choose an alarm type).
- Don't make your alarm's trigger time any more precise than it has to be.

  Use `setInexactRepeating()` instead of `setRepeating()`. 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 Android 4.4 (API Level 19), all repeating alarms are inexact. Note that while `setInexactRepeating()` is an improvement over `setRepeating()`, it can still overwhelm a server if every instance of an app hits the server around the same time. Therefore, for network requests, add some randomness to your alarms, as discussed above.

- Avoid basing your alarm on clock time if possible.

  Repeating alarms that are based on a precise trigger time don't scale well. Use `ELAPSED_REALTIME` if you can. The different alarm types are described in more detail in the following section.

Set a Repeating Alarm

As described above, repeating alarms are a good choice for scheduling

regular events or data lookups. A repeating alarm has the following characteristics:

- A alarm type. For more discussion, see Choose an alarm type.
- A trigger time. If the trigger time you specify is in the past, the alarm triggers immediately.
- The alarm's interval. For example, once a day, every hour, every 5 seconds, and so on.
- A pending intent that fires when the alarm is triggered. When you set a second alarm that uses the same pending intent, it replaces the original alarm.

Choose an alarm type

One of the first considerations in using a repeating alarm is what its type should be.

There are two general clock types for alarms: "elapsed real time" and "real time clock" (RTC). 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 clock-based real time clock types. Note, however, that 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. We recommend that you use a "elapsed real time" alarm if you can.

Here is the list of types:

- ELAPSED_REALTIME—Fires the pending intent based on the amount of time since the device was booted, but doesn't wake up the device. The elapsed time includes any time during which the device was asleep.
- ELAPSED_REALTIME_WAKEUP—Wakes up the device and fires the pending intent after the specified length of time has elapsed since device boot.
- RTC—Fires the pending intent at the specified time but does not wake up the device.
- RTC_WAKEUP—Wakes up the device to fire the pending intent at the specified time.

ELAPSED_REALTIME_WAKEUP examples

Here are some examples of using ELAPSED_REALTIME_WAKEUP.

Wake up the device to fire the alarm in 30 minutes, and every 30 minutes after that:

```
// Hopefully your alarm will have a lower frequency than this!
alarmMgr.setInexactRepeating(AlarmManager.ELAPSED_REALTIME_WAKEUP,
        AlarmManager.INTERVAL_HALF_HOUR,
        AlarmManager.INTERVAL_HALF_HOUR, alarmIntent);
```

Wake up the device to fire a one-time (non-repeating) alarm in one minute:

```
private AlarmManager alarmMgr;
private PendingIntent alarmIntent;
...
alarmMgr =
(AlarmManager)context.getSystemService(Context.ALARM_SERVICE);
Intent intent = new Intent(context, AlarmReceiver.class);
alarmIntent = PendingIntent.getBroadcast(context, 0, intent, 0);
```

```
alarmMgr.set(AlarmManager.ELAPSED_REALTIME_WAKEUP,
        SystemClock.elapsedRealtime() +
        60 * 1000, alarmIntent);
```

RTC examples

Here are some examples of using RTC_WAKEUP.

Wake up the device to fire the alarm at approximately 2:00 p.m., and repeat once a day at the same time:

```
// Set the alarm to start at approximately 2:00 p.m.
Calendar calendar = Calendar.getInstance();
calendar.setTimeInMillis(System.currentTimeMillis());
calendar.set(Calendar.HOUR_OF_DAY, 14);

// With setInexactRepeating(), you have to use one of the
AlarmManager interval
// constants--in this case, AlarmManager.INTERVAL_DAY.
alarmMgr.setInexactRepeating(AlarmManager.RTC_WAKEUP,
calendar.getTimeInMillis(),
        AlarmManager.INTERVAL_DAY, alarmIntent);
```

Wake up the device to fire the alarm at precisely 8:30 a.m., and every 20 minutes thereafter:

```
private AlarmManager alarmMgr;
private PendingIntent alarmIntent;
...
alarmMgr =
(AlarmManager)context.getSystemService(Context.ALARM_SERVICE);
Intent intent = new Intent(context, AlarmReceiver.class);
alarmIntent = PendingIntent.getBroadcast(context, 0, intent, 0);

// Set the alarm to start at 8:30 a.m.
Calendar calendar = Calendar.getInstance();
calendar.setTimeInMillis(System.currentTimeMillis());
calendar.set(Calendar.HOUR_OF_DAY, 8);
calendar.set(Calendar.MINUTE, 30);

// setRepeating() lets you specify a precise custom interval--in
this case,
// 20 minutes.
alarmMgr.setRepeating(AlarmManager.RTC_WAKEUP,
calendar.getTimeInMillis(),
        1000 * 60 * 20, alarmIntent);
```

Decide how precise your alarm needs to be

As described above, choosing the alarm type is often the first step in creating an alarm. A further distinction is how precise you need your

alarm to be. For most apps, `setInexactRepeating()` is the right choice. When you use this method, Android synchronizes multiple inexact repeating alarms and fires them at the same time. This reduces the drain on the battery.

For the rare app that has rigid time requirements—for example, the alarm needs to fire precisely at 8:30 a.m., and every hour on the hour thereafter—use `setRepeating()`. But you should avoid using exact alarms if possible.

With `setInexactRepeating()`, you can't specify a custom interval the way you can with `setRepeating()`. You have to use one of the interval constants, such as `INTERVAL_FIFTEEN_MINUTES`, `INTERVAL_DAY`, and so on. See `AlarmManager` for the complete list.

Cancel an Alarm

Depending on your app, you may want to include the ability to cancel the alarm. To cancel an alarm, call `cancel()` on the Alarm Manager, passing in the `PendingIntent` you no longer want to fire. For example:

```
// If the alarm has been set, cancel it.
if (alarmMgr!= null) {
    alarmMgr.cancel(alarmIntent);
}
```

Start an Alarm When the Device Boots

By default, all alarms are canceled when a device shuts down. To prevent this from happening, you can design your application to automatically restart a repeating alarm if the user reboots the device. This ensures that the `AlarmManager` will continue doing its task without the user needing to manually restart the alarm.

Here are the steps:

1. Set the `RECEIVE_BOOT_COMPLETED` permission in your application's manifest. This allows your app to receive the `ACTION_BOOT_COMPLETED` that is broadcast after the system finishes booting (this only works if

the app has already been launched by the user at least once):

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

2. Implement a `BroadcastReceiver` to receive the broadcast:

```
public class SampleBootReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        if
(intent.getAction().equals("android.intent.action.BOOT_COMPLETED")
{
            // Set the alarm here.
        }
    }
}
```

3. Add the receiver to your app's manifest file with an intent filter that filters on the `ACTION_BOOT_COMPLETED` action:

```
<receiver android:name=".SampleBootReceiver"
        android:enabled="false">
    <intent-filter>
        <action
android:name="android.intent.action.BOOT_COMPLETED"></action>
    </intent-filter>
</receiver>
```

Notice that in the manifest, the boot receiver is set to `android:enabled="false"`. This means that the receiver will not be called unless the application explicitly enables it. This prevents the boot receiver from being called unnecessarily. You can enable a receiver (for example, if the user sets an alarm) as follows:

```
ComponentName receiver = new ComponentName(context,
SampleBootReceiver.class);
PackageManager pm = context.getPackageManager();

pm.setComponentEnabledSetting(receiver,
        PackageManager.COMPONENT_ENABLED_STATE_ENABLED,
        PackageManager.DONT_KILL_APP);
```

Once you enable the receiver this way, it will stay enabled, even if the user reboots the device. In other words, programmatically enabling the receiver overrides the manifest setting, even across reboots. The receiver will stay enabled until your app disables it. You can disable a receiver (for example, if the user cancels an alarm) as follows:

```
ComponentName receiver = new ComponentName(context,
SampleBootReceiver.class);
PackageManager pm = context.getPackageManager();

pm.setComponentEnabledSetting(receiver,
        PackageManager.COMPONENT_ENABLED_STATE_DISABLED,
        PackageManager.DONT_KILL_APP);
```

Next Get started Next class

Best Practices for Performance

These classes and articles help you build an app that's smooth, responsive, and uses as little battery as possible.

Managing Your App's Memory

In this document

1. How Android Manages Memory
    1. Sharing Memory
    2. Allocating and Reclaiming App Memory
    3. Restricting App Memory
    4. Switching Apps
2. How Your App Should Manage Memory
    1. Use services sparingly
    2. Release memory when your user interface becomes hidden
    3. Release memory as memory becomes tight
    4. Check how much memory you should use
    5. Avoid wasting memory with bitmaps
    6. Use optimized data containers
    7. Be aware of memory overhead
    8. Be careful with code abstractions
    9. Use nano protobufs for serialized data
    10. Avoid dependency injection frameworks
    11. Be careful about using external libraries
    12. Optimize overall performance
    13. Use ProGuard to strip out any unneeded code
    14. Use zipalign on your final APK
    15. Analyze your RAM usage
    16. Use multiple processes

See Also

- Investigating Your RAM Usage

Random-access memory (RAM) is a valuable resource in any software development environment, but it's even more valuable on a mobile

operating system where physical memory is often constrained. Although Android's Dalvik virtual machine performs routine garbage collection, this doesn't allow you to ignore when and where your app allocates and releases memory.

In order for the garbage collector to reclaim memory from your app, you need to avoid introducing memory leaks (usually caused by holding onto object references in global members) and release any `Reference` objects at the appropriate time (as defined by lifecycle callbacks discussed further below). For most apps, the Dalvik garbage collector takes care of the rest: the system reclaims your memory allocations when the corresponding objects leave the scope of your app's active threads.

This document explains how Android manages app processes and memory allocation, and how you can proactively reduce memory usage while developing for Android. For more information about general practices to clean up your resources when programming in Java, refer to other books or online documentation about managing resource references. If you're looking for information about how to analyze your app's memory once you've already built it, read Investigating Your RAM Usage.

How Android Manages Memory

Android does not offer swap space for memory, but it does use paging and memory-mapping (mmapping) to manage memory. This means that any memory you modify—whether by allocating new objects or touching mmapped pages—remains resident in RAM and cannot be paged out. So the only way to completely release memory from your app is to release object references you may be holding, making the memory available to the garbage collector. That is with one exception: any files mmapped in without modification, such as code, can be paged out of RAM if the system wants to use that memory elsewhere.

Sharing Memory

In order to fit everything it needs in RAM, Android tries to share RAM

pages across processes. It can do so in the following ways:

- Each app process is forked from an existing process called Zygote. The Zygote process starts when the system boots and loads common framework code and resources (such as activity themes). To start a new app process, the system forks the Zygote process then loads and runs the app's code in the new process. This allows most of the RAM pages allocated for framework code and resources to be shared across all app processes.
- Most static data is mmapped into a process. This not only allows that same data to be shared between processes but also allows it to be paged out when needed. Example static data include: Dalvik code (by placing it in a pre-linked `.odex` file for direct mmapping), app resources (by designing the resource table to be a structure that can be mmapped and by aligning the zip entries of the APK), and traditional project elements like native code in `.so` files.
- In many places, Android shares the same dynamic RAM across processes using explicitly allocated shared memory regions (either with ashmem or gralloc). For example, window surfaces use shared memory between the app and screen compositor, and cursor buffers use shared memory between the content provider and client.

Due to the extensive use of shared memory, determining how much memory your app is using requires care. Techniques to properly determine your app's memory use are discussed in Investigating Your RAM Usage.

Allocating and Reclaiming App Memory

Here are some facts about how Android allocates then reclaims memory from your app:

- The Dalvik heap for each process is constrained to a single virtual memory range. This defines the logical heap size, which can grow as it needs to (but only up to a limit that the system defines for each app).
- The logical size of the heap is not the same as the amount of physical memory used by the heap. When inspecting your app's

heap, Android computes a value called the Proportional Set Size (PSS), which accounts for both dirty and clean pages that are shared with other processes—but only in an amount that's proportional to how many apps share that RAM. This (PSS) total is what the system considers to be your physical memory footprint. For more information about PSS, see the Investigating Your RAM Usage guide.
- The Dalvik heap does not compact the logical size of the heap, meaning that Android does not defragment the heap to close up space. Android can only shrink the logical heap size when there is unused space at the end of the heap. But this doesn't mean the physical memory used by the heap can't shrink. After garbage collection, Dalvik walks the heap and finds unused pages, then returns those pages to the kernel using madvise. So, paired allocations and deallocations of large chunks should result in reclaiming all (or nearly all) the physical memory used. However, reclaiming memory from small allocations can be much less efficient because the page used for a small allocation may still be shared with something else that has not yet been freed.

Restricting App Memory

To maintain a functional multi-tasking environment, Android sets a hard limit on the heap size for each app. The exact heap size limit varies between devices based on how much RAM the device has available overall. If your app has reached the heap capacity and tries to allocate more memory, it will receive an `OutOfMemoryError`.

In some cases, you might want to query the system to determine exactly how much heap space you have available on the current device—for example, to determine how much data is safe to keep in a cache. You can query the system for this figure by calling `getMemoryClass()`. This returns an integer indicating the number of megabytes available for your app's heap. This is discussed further below, under Check how much memory you should use.

Switching Apps

Instead of using swap space when the user switches between apps, Android keeps processes that are not hosting a foreground ("user visible") app component in a least-recently used (LRU) cache. For example, when the user first launches an app, a process is created for it, but when the user leaves the app, that process does *not* quit. The system keeps the process cached, so if the user later returns to the app, the process is reused for faster app switching.

If your app has a cached process and it retains memory that it currently does not need, then your app—even while the user is not using it—is constraining the system's overall performance. So, as the system runs low on memory, it may kill processes in the LRU cache beginning with the process least recently used, but also giving some consideration toward which processes are most memory intensive. To keep your process cached as long as possible, follow the advice in the following sections about when to release your references.

More information about how processes are cached while not running in the foreground and how Android decides which ones can be killed is available in the Processes and Threads guide.

How Your App Should Manage Memory

You should consider RAM constraints throughout all phases of development, including during app design (before you begin development). There are many ways you can design and write code that lead to more efficient results, through aggregation of the same techniques applied over and over.

You should apply the following techniques while designing and implementing your app to make it more memory efficient.

Use services sparingly

If your app needs a service to perform work in the background, do not keep it running unless it's actively performing a job. Also be careful to never leak your service by failing to stop it when its work is done.

When you start a service, the system prefers to always keep the process for that service running. This makes the process very expensive because the RAM used by the service can't be used by anything else or paged out. This reduces the number of cached processes that the system can keep in the LRU cache, making app switching less efficient. It can even lead to thrashing in the system when memory is tight and the system can't maintain enough processes to host all the services currently running.

The best way to limit the lifespan of your service is to use an `IntentService`, which finishes itself as soon as it's done handling the intent that started it. For more information, read Running in a Background Service .

Leaving a service running when it's not needed is **one of the worst memory-management mistakes** an Android app can make. So don't be greedy by keeping a service for your app running. Not only will it increase the risk of your app performing poorly due to RAM constraints, but users will discover such misbehaving apps and uninstall them.

Release memory when your user interface becomes hidden

When the user navigates to a different app and your UI is no longer visible, you should release any resources that are used by only your UI. Releasing UI resources at this time can significantly increase the system's capacity for cached processes, which has a direct impact on the quality of the user experience.

To be notified when the user exits your UI, implement the `onTrimMemory()` callback in your `Activity` classes. You should use this method to listen for the `TRIM_MEMORY_UI_HIDDEN` level, which indicates your UI is now hidden from view and you should free resources that only your UI uses.

Notice that your app receives the `onTrimMemory()` callback with `TRIM_MEMORY_UI_HIDDEN` only when *all the UI components* of your app process become hidden from the user. This is distinct from the `onStop()`

callback, which is called when an `Activity` instance becomes hidden, which occurs even when the user moves to another activity in your app. So although you should implement `onStop()` to release activity resources such as a network connection or to unregister broadcast receivers, you usually should not release your UI resources until you receive `onTrimMemory(TRIM_MEMORY_UI_HIDDEN)`. This ensures that if the user navigates *back* from another activity in your app, your UI resources are still available to resume the activity quickly.

Release memory as memory becomes tight

During any stage of your app's lifecycle, the `onTrimMemory()` callback also tells you when the overall device memory is getting low. You should respond by further releasing resources based on the following memory levels delivered by `onTrimMemory()`:

- `TRIM_MEMORY_RUNNING_MODERATE`

  Your app is running and not considered killable, but the device is running low on memory and the system is actively killing processes in the LRU cache.

- `TRIM_MEMORY_RUNNING_LOW`

  Your app is running and not considered killable, but the device is running much lower on memory so you should release unused resources to improve system performance (which directly impacts your app's performance).

- `TRIM_MEMORY_RUNNING_CRITICAL`

  Your app is still running, but the system has already killed most of the processes in the LRU cache, so you should release all non-critical resources now. If the system cannot reclaim sufficient amounts of RAM, it will clear all of the LRU cache and begin killing processes that the system prefers to keep alive, such as those hosting a running service.

Also, when your app process is currently cached, you may receive one of the following levels from `onTrimMemory()`:

- `TRIM_MEMORY_BACKGROUND`

  The system is running low on memory and your process is near the beginning of the LRU list. Although your app process is not at a high risk of being killed, the system may already be killing processes in the LRU cache. You should release resources that are easy to recover so your process will remain in the list and resume quickly when the user returns to your app.

- `TRIM_MEMORY_MODERATE`

  The system is running low on memory and your process is near the middle of the LRU list. If the system becomes further constrained for memory, there's a chance your process will be killed.

- `TRIM_MEMORY_COMPLETE`

  The system is running low on memory and your process is one of the first to be killed if the system does not recover memory now. You should release everything that's not critical to resuming your app state.

Because the `onTrimMemory()` callback was added in API level 14, you can use the `onLowMemory()` callback as a fallback for older versions, which is roughly equivalent to the `TRIM_MEMORY_COMPLETE` event.

Note: When the system begins killing processes in the LRU cache, although it primarily works bottom-up, it does give some consideration to which processes are consuming more memory and will thus provide the system more memory gain if killed. So the less memory you consume while in the LRU list overall, the better your chances are to remain in the list and be able to quickly resume.

Check how much memory you should use

As mentioned earlier, each Android-powered device has a different

amount of RAM available to the system and thus provides a different heap limit for each app. You can call `getMemoryClass()` to get an estimate of your app's available heap in megabytes. If your app tries to allocate more memory than is available here, it will receive an `OutOfMemoryError`.

In very special situations, you can request a larger heap size by setting the `largeHeap` attribute to "true" in the manifest `<application>` tag. If you do so, you can call `getLargeMemoryClass()` to get an estimate of the large heap size.

However, the ability to request a large heap is intended only for a small set of apps that can justify the need to consume more RAM (such as a large photo editing app). **Never request a large heap simply because you've run out of memory** and you need a quick fix—you should use it only when you know exactly where all your memory is being allocated and why it must be retained. Yet, even when you're confident your app can justify the large heap, you should avoid requesting it to whatever extent possible. Using the extra memory will increasingly be to the detriment of the overall user experience because garbage collection will take longer and system performance may be slower when task switching or performing other common operations.

Additionally, the large heap size is not the same on all devices and, when running on devices that have limited RAM, the large heap size may be exactly the same as the regular heap size. So even if you do request the large heap size, you should call `getMemoryClass()` to check the regular heap size and strive to always stay below that limit.

Avoid wasting memory with bitmaps

When you load a bitmap, keep it in RAM only at the resolution you need for the current device's screen, scaling it down if the original bitmap is a higher resolution. Keep in mind that an increase in bitmap resolution results in a corresponding (increase^2) in memory needed, because both the X and Y dimensions increase.

Note: On Android 2.3.x (API level 10) and below, bitmap objects always appear as the same size in your app heap regardless of the image

resolution (the actual pixel data is stored separately in native memory). This makes it more difficult to debug the bitmap memory allocation because most heap analysis tools do not see the native allocation. However, beginning in Android 3.0 (API level 11), the bitmap pixel data is allocated in your app's Dalvik heap, improving garbage collection and debuggability. So if your app uses bitmaps and you're having trouble discovering why your app is using some memory on an older device, switch to a device running Android 3.0 or higher to debug it.

For more tips about working with bitmaps, read Managing Bitmap Memory.

Use optimized data containers

Take advantage of optimized containers in the Android framework, such as `SparseArray`, `SparseBooleanArray`, and `LongSparseArray`. The generic `HashMap` implementation can be quite memory inefficient because it needs a separate entry object for every mapping. Additionally, the `SparseArray` classes are more efficient because they avoid the system's need to autobox the key and sometimes value (which creates yet another object or two per entry). And don't be afraid of dropping down to raw arrays when that makes sense.

Be aware of memory overhead

Be knowledgeable about the cost and overhead of the language and libraries you are using, and keep this information in mind when you design your app, from start to finish. Often, things on the surface that look innocuous may in fact have a large amount of overhead. Examples include:

- Enums often require more than twice as much memory as static constants. You should strictly avoid using enums on Android.
- Every class in Java (including anonymous inner classes) uses about 500 bytes of code.
- Every class instance has 12-16 bytes of RAM overhead.
- Putting a single entry into a `HashMap` requires the allocation of an

additional entry object that takes 32 bytes (see the previous section about optimized data containers).

A few bytes here and there quickly add up—app designs that are class- or object-heavy will suffer from this overhead. That can leave you in the difficult position of looking at a heap analysis and realizing your problem is a lot of small objects using up your RAM.

Be careful with code abstractions

Often, developers use abstractions simply as a "good programming practice," because abstractions can improve code flexibility and maintenance. However, abstractions come at a significant cost: generally they require a fair amount more code that needs to be executed, requiring more time and more RAM for that code to be mapped into memory. So if your abstractions aren't supplying a significant benefit, you should avoid them.

Use nano protobufs for serialized data

Protocol buffers are a language-neutral, platform-neutral, extensible mechanism designed by Google for serializing structured data—think XML, but smaller, faster, and simpler. If you decide to use protobufs for your data, you should always use nano protobufs in your client-side code. Regular protobufs generate extremely verbose code, which will cause many kinds of problems in your app: increased RAM use, significant APK size increase, slower execution, and quickly hitting the DEX symbol limit.

For more information, see the "Nano version" section in the protobuf readme.

Avoid dependency injection frameworks

Using a dependency injection framework such as Guice or RoboGuice may be attractive because they can simplify the code you write and provide an adaptive environment that's useful for testing and other

configuration changes. However, these frameworks tend to perform a lot of process initialization by scanning your code for annotations, which can require significant amounts of your code to be mapped into RAM even though you don't need it. These mapped pages are allocated into clean memory so Android can drop them, but that won't happen until the pages have been left in memory for a long period of time.

Be careful about using external libraries

External library code is often not written for mobile environments and can be inefficient when used for work on a mobile client. At the very least, when you decide to use an external library, you should assume you are taking on a significant porting and maintenance burden to optimize the library for mobile. Plan for that work up-front and analyze the library in terms of code size and RAM footprint before deciding to use it at all.

Even libraries supposedly designed for use on Android are potentially dangerous because each library may do things differently. For example, one library may use nano protobufs while another uses micro protobufs. Now you have two different protobuf implementations in your app. This can and will also happen with different implementations of logging, analytics, image loading frameworks, caching, and all kinds of other things you don't expect. ProGuard won't save you here because these will all be lower-level dependencies that are required by the features for which you want the library. This becomes especially problematic when you use an `Activity` subclass from a library (which will tend to have wide swaths of dependencies), when libraries use reflection (which is common and means you need to spend a lot of time manually tweaking ProGuard to get it to work), and so on.

Also be careful not to fall into the trap of using a shared library for one or two features out of dozens of other things it does; you don't want to pull in a large amount of code and overhead that you don't even use. At the end of the day, if there isn't an existing implementation that is a strong match for what you need to do, it may be best if you create your own implementation.

Optimize overall performance

A variety of information about optimizing your app's overall performance is available in other documents listed in Best Practices for Performance. Many of these documents include optimizations tips for CPU performance, but many of these tips also help optimize your app's memory use, such as by reducing the number of layout objects required by your UI.

You should also read about optimizing your UI with the layout debugging tools and take advantage of the optimization suggestions provided by the lint tool.

Use ProGuard to strip out any unneeded code

The ProGuard tool shrinks, optimizes, and obfuscates your code by removing unused code and renaming classes, fields, and methods with semantically obscure names. Using ProGuard can make your code more compact, requiring fewer RAM pages to be mapped.

Use zipalign on your final APK

If you do any post-processing of an APK generated by a build system (including signing it with your final production certificate), then you must run zipalign on it to have it re-aligned. Failing to do so can cause your app to require significantly more RAM, because things like resources can no longer be mmapped from the APK.

Note: Google Play Store does not accept APK files that are not zipaligned.

Analyze your RAM usage

Once you achieve a relatively stable build, begin analyzing how much RAM your app is using throughout all stages of its lifecycle. For information about how to analyze your app, read Investigating Your RAM Usage.

Use multiple processes

If it's appropriate for your app, an advanced technique that may help you manage your app's memory is dividing components of your app into multiple processes. This technique must always be used carefully and **most apps should not run multiple processes**, as it can easily increase—rather than decrease—your RAM footprint if done incorrectly. It is primarily useful to apps that may run significant work in the background as well as the foreground and can manage those operations separately.

An example of when multiple processes may be appropriate is when building a music player that plays music from a service for long period of time. If the entire app runs in one process, then many of the allocations performed for its activity UI must be kept around as long as it is playing music, even if the user is currently in another app and the service is controlling the playback. An app like this may be split into two process: one for its UI, and the other for the work that continues running in the background service.

You can specify a separate process for each app component by declaring the `android:process` attribute for each component in the manifest file. For example, you can specify that your service should run in a process separate from your app's main process by declaring a new process named "background" (but you can name the process anything you like):

```
<service android:name=".PlaybackService"
         android:process=":background" />
```

Your process name should begin with a colon (':') to ensure that the process remains private to your app.

Before you decide to create a new process, you need to understand the memory implications. To illustrate the consequences of each process, consider that an empty process doing basically nothing has an extra memory footprint of about 1.4MB, as shown by the memory information dump below.

```
adb shell dumpsys meminfo com.example.android.apis:empty
```

```
** MEMINFO in pid 10172 [com.example.android.apis:empty] **
                 Pss      Pss  Shared Private  Shared Private    Heap
Heap    Heap
               Total    Clean    Dirty   Dirty    Clean   Clean    Size
Alloc    Free
              ------   ------   ------  ------   ------  ------  ----
--  ------  ------
  Native Heap     0        0        0       0        0       0    1864
1800      63
  Dalvik Heap   764        0     5228     316        0       0    5584
5499      85
 Dalvik Other   619        0     3784     448        0       0
        Stack    28        0        8      28        0       0
    Other dev     4        0       12       0        0       4
     .so mmap   287        0     2840     212      972       0
    .apk mmap    54        0        0       0      136       0
    .dex mmap   250      148        0       0     3704     148
   Other mmap     8        0        8       8       20       0
      Unknown   403        0      600     380        0       0
        TOTAL  2417      148    12480    1392     4832     152    7448
7299     148
```

Note: More information about how to read this output is provided in Investigating Your RAM Usage. The key data here is the *Private Dirty* and *Private Clean* memory, which shows that this process is using almost 1.4MB of non-pageable RAM (distributed across the Dalvik heap, native allocations, book-keeping, and library-loading), and another 150K of RAM for code that has been mapped in to execute.

This memory footprint for an empty process is fairly significant and it can quickly grow as you start doing work in that process. For example, here is the memory use of a process that is created only to show an activity with some text in it:

```
** MEMINFO in pid 10226 [com.example.android.helloactivity] **
                 Pss      Pss  Shared Private  Shared Private    Heap
Heap    Heap
               Total    Clean    Dirty   Dirty    Clean   Clean    Size
Alloc    Free
              ------   ------   ------  ------   ------  ------  ----
--  ------  ------
  Native Heap     0        0        0       0        0       0    3000
2951      48
  Dalvik Heap  1074        0     4928     776        0       0    5744
5658      86
 Dalvik Other   802        0     3612     664        0       0
        Stack    28        0        8      28        0       0
       Ashmem     6        0       16       0        0       0
    Other dev   108        0       24     104        0       4
     .so mmap  2166        0     2824    1828     3756       0
    .apk mmap    48        0        0       0      632       0
    .ttf mmap     3        0        0       0       24       0
    .dex mmap   292        4        0       0     5672       4
```

```
   Other mmap    10        0        8        8       68        0
      Unknown   632        0      412      624        0        0
        TOTAL  5169        4    11832     4032    10152        8     8744
8609     134
```

The process has now almost tripled in size, to 4MB, simply by showing some text in the UI. This leads to an important conclusion: If you are going to split your app into multiple processes, only one process should be responsible for UI. Other processes should avoid any UI, as this will quickly increase the RAM required by the process (especially once you start loading bitmap assets and other resources). It may then be hard or impossible to reduce the memory usage once the UI is drawn.

Additionally, when running more than one process, it's more important than ever that you keep your code as lean as possible, because any unnecessary RAM overhead for common implementations are now replicated in each process. For example, if you are using enums (though you should not use enums), all of the RAM needed to create and initialize those constants is duplicated in each process, and any abstractions you have with adapters and temporaries or other overhead will likewise be replicated.

Another concern with multiple processes is the dependencies that exist between them. For example, if your app has a content provider that you have running in the default process which also hosts your UI, then code in a background process that uses that content provider will also require that your UI process remain in RAM. If your goal is to have a background process that can run independently of a heavy-weight UI process, it can't have dependencies on content providers or services that execute in the UI process.

Next Get started Next class

Performance Tips

In this document

1. Avoid Creating Unnecessary Objects
2. Prefer Static Over Virtual
3. Use Static Final For Constants
4. Avoid Internal Getters/Setters
5. Use Enhanced For Loop Syntax
6. Consider Package Instead of Private Access with Private Inner Classes
7. Avoid Using Floating-Point
8. Know and Use the Libraries
9. Use Native Methods Carefully
10. Use Native Methods Judiciously
11. Closing Notes

This document primarily covers micro-optimizations that can improve overall app performance when combined, but it's unlikely that these changes will result in dramatic performance effects. Choosing the right algorithms and data structures should always be your priority, but is outside the scope of this document. You should use the tips in this document as general coding practices that you can incorporate into your habits for general code efficiency.

There are two basic rules for writing efficient code:

- Don't do work that you don't need to do.
- Don't allocate memory if you can avoid it.

One of the trickiest problems you'll face when micro-optimizing an Android app is that your app is certain to be running on multiple types of hardware. Different versions of the VM running on different processors running at different speeds. It's not even generally the case that you can

simply say "device X is a factor F faster/slower than device Y", and scale your results from one device to others. In particular, measurement on the emulator tells you very little about performance on any device. There are also huge differences between devices with and without a JIT: the best code for a device with a JIT is not always the best code for a device without.

To ensure your app performs well across a wide variety of devices, ensure your code is efficient at all levels and agressively optimize your performance.

Avoid Creating Unnecessary Objects

Object creation is never free. A generational garbage collector with per-thread allocation pools for temporary objects can make allocation cheaper, but allocating memory is always more expensive than not allocating memory.

As you allocate more objects in your app, you will force a periodic garbage collection, creating little "hiccups" in the user experience. The concurrent garbage collector introduced in Android 2.3 helps, but unnecessary work should always be avoided.

Thus, you should avoid creating object instances you don't need to. Some examples of things that can help:

- If you have a method returning a string, and you know that its result will always be appended to a `StringBuffer` anyway, change your signature and implementation so that the function does the append directly, instead of creating a short-lived temporary object.
- When extracting strings from a set of input data, try to return a substring of the original data, instead of creating a copy. You will create a new `String` object, but it will share the `char[]` with the data. (The trade-off being that if you're only using a small part of the original input, you'll be keeping it all around in memory anyway if you go this route.)

A somewhat more radical idea is to slice up multidimensional arrays into

parallel single one-dimension arrays:

- An array of `int`s is a much better than an array of `Integer` objects, but this also generalizes to the fact that two parallel arrays of ints are also a **lot** more efficient than an array of `(int,int)` objects. The same goes for any combination of primitive types.
- If you need to implement a container that stores tuples of `(Foo,Bar)` objects, try to remember that two parallel `Foo[]` and `Bar[]` arrays are generally much better than a single array of custom `(Foo,Bar)` objects. (The exception to this, of course, is when you're designing an API for other code to access. In those cases, it's usually better to make a small compromise to the speed in order to achieve a good API design. But in your own internal code, you should try and be as efficient as possible.)

Generally speaking, avoid creating short-term temporary objects if you can. Fewer objects created mean less-frequent garbage collection, which has a direct impact on user experience.

Prefer Static Over Virtual

If you don't need to access an object's fields, make your method static. Invocations will be about 15%-20% faster. It's also good practice, because you can tell from the method signature that calling the method can't alter the object's state.

Use Static Final For Constants

Consider the following declaration at the top of a class:

```
static int intVal = 42;
static String strVal = "Hello, world!";
```

The compiler generates a class initializer method, called `<clinit>`, that is executed when the class is first used. The method stores the value 42 into `intVal`, and extracts a reference from the classfile string constant table for `strVal`. When these values are referenced later on, they are accessed with field lookups.

We can improve matters with the "final" keyword:

```
static final int intVal = 42;
static final String strVal = "Hello, world!";
```

The class no longer requires a `<clinit>` method, because the constants go into static field initializers in the dex file. Code that refers to `intVal` will use the integer value 42 directly, and accesses to `strVal` will use a relatively inexpensive "string constant" instruction instead of a field lookup.

Note: This optimization applies only to primitive types and `String` constants, not arbitrary reference types. Still, it's good practice to declare constants `static final` whenever possible.

Avoid Internal Getters/Setters

In native languages like C++ it's common practice to use getters (`i = getCount()`) instead of accessing the field directly (`i = mCount`). This is an excellent habit for C++ and is often practiced in other object oriented languages like C# and Java, because the compiler can usually inline the access, and if you need to restrict or debug field access you can add the code at any time.

However, this is a bad idea on Android. Virtual method calls are expensive, much more so than instance field lookups. It's reasonable to follow common object-oriented programming practices and have getters and setters in the public interface, but within a class you should always access fields directly.

Without a JIT, direct field access is about 3x faster than invoking a trivial getter. With the JIT (where direct field access is as cheap as accessing a local), direct field access is about 7x faster than invoking a trivial getter.

Note that if you're using ProGuard, you can have the best of both worlds because ProGuard can inline accessors for you.

Use Enhanced For Loop Syntax

The enhanced `for` loop (also sometimes known as "for-each" loop) can be used for collections that implement the `Iterable` interface and for arrays. With collections, an iterator is allocated to make interface calls to `hasNext()` and `next()`. With an `ArrayList`, a hand-written counted loop is about 3x faster (with or without JIT), but for other collections the enhanced for loop syntax will be exactly equivalent to explicit iterator usage.

There are several alternatives for iterating through an array:

```
static class Foo {
    int mSplat;
}

Foo[] mArray = ...

public void zero() {
    int sum = 0;
    for (int i = 0; i < mArray.length; ++i) {
        sum += mArray[i].mSplat;
    }
}

public void one() {
    int sum = 0;
    Foo[] localArray = mArray;
    int len = localArray.length;

    for (int i = 0; i < len; ++i) {
        sum += localArray[i].mSplat;
    }
}

public void two() {
    int sum = 0;
    for (Foo a : mArray) {
        sum += a.mSplat;
    }
}
```

`zero()` is slowest, because the JIT can't yet optimize away the cost of getting the array length once for every iteration through the loop.

`one()` is faster. It pulls everything out into local variables, avoiding the lookups. Only the array length offers a performance benefit.

`two()` is fastest for devices without a JIT, and indistinguishable from **one()** for devices with a JIT. It uses the enhanced for loop syntax

introduced in version 1.5 of the Java programming language.

So, you should use the enhanced `for` loop by default, but consider a hand-written counted loop for performance-critical `ArrayList` iteration.

Tip: Also see Josh Bloch's *Effective Java*, item 46.

Consider Package Instead of Private Access with Private Inner Classes

Consider the following class definition:

```
public class Foo {
    private class Inner {
        void stuff() {
            Foo.this.doStuff(Foo.this.mValue);
        }
    }

    private int mValue;

    public void run() {
        Inner in = new Inner();
        mValue = 27;
        in.stuff();
    }

    private void doStuff(int value) {
        System.out.println("Value is " + value);
    }
}
```

What's important here is that we define a private inner class (`Foo$Inner`) that directly accesses a private method and a private instance field in the outer class. This is legal, and the code prints "Value is 27" as expected.

The problem is that the VM considers direct access to `Foo`'s private members from `Foo$Inner` to be illegal because `Foo` and `Foo$Inner` are different classes, even though the Java language allows an inner class to access an outer class' private members. To bridge the gap, the compiler generates a couple of synthetic methods:

```
/*package*/ static int Foo.access$100(Foo foo) {
    return foo.mValue;
}
/*package*/ static void Foo.access$200(Foo foo, int value) {
    foo.doStuff(value);
```

```
}
```

The inner class code calls these static methods whenever it needs to access the `mValue` field or invoke the `doStuff()` method in the outer class. What this means is that the code above really boils down to a case where you're accessing member fields through accessor methods. Earlier we talked about how accessors are slower than direct field accesses, so this is an example of a certain language idiom resulting in an "invisible" performance hit.

If you're using code like this in a performance hotspot, you can avoid the overhead by declaring fields and methods accessed by inner classes to have package access, rather than private access. Unfortunately this means the fields can be accessed directly by other classes in the same package, so you shouldn't use this in public API.

Avoid Using Floating-Point

As a rule of thumb, floating-point is about 2x slower than integer on Android-powered devices.

In speed terms, there's no difference between `float` and `double` on the more modern hardware. Space-wise, `double` is 2x larger. As with desktop machines, assuming space isn't an issue, you should prefer `double` to `float`.

Also, even for integers, some processors have hardware multiply but lack hardware divide. In such cases, integer division and modulus operations are performed in software—something to think about if you're designing a hash table or doing lots of math.

Know and Use the Libraries

In addition to all the usual reasons to prefer library code over rolling your own, bear in mind that the system is at liberty to replace calls to library methods with hand-coded assembler, which may be better than the best code the JIT can produce for the equivalent Java. The typical example

here is `String.indexOf()` and related APIs, which Dalvik replaces with an inlined intrinsic. Similarly, the `System.arraycopy()` method is about 9x faster than a hand-coded loop on a Nexus One with the JIT.

Tip: Also see Josh Bloch's *Effective Java*, item 47.

Use Native Methods Carefully

Developing your app with native code using the Android NDK isn't necessarily more efficient than programming with the Java language. For one thing, there's a cost associated with the Java-native transition, and the JIT can't optimize across these boundaries. If you're allocating native resources (memory on the native heap, file descriptors, or whatever), it can be significantly more difficult to arrange timely collection of these resources. You also need to compile your code for each architecture you wish to run on (rather than rely on it having a JIT). You may even have to compile multiple versions for what you consider the same architecture: native code compiled for the ARM processor in the G1 can't take full advantage of the ARM in the Nexus One, and code compiled for the ARM in the Nexus One won't run on the ARM in the G1.

Native code is primarily useful when you have an existing native codebase that you want to port to Android, not for "speeding up" parts of your Android app written with the Java language.

If you do need to use native code, you should read our JNI Tips.

Tip: Also see Josh Bloch's *Effective Java*, item 54.

Performance Myths

On devices without a JIT, it is true that invoking methods via a variable with an exact type rather than an interface is slightly more efficient. (So, for example, it was cheaper to invoke methods on a `HashMap map` than a `Map map`, even though in both cases the map was a `HashMap`.) It was not the case that this was 2x slower; the actual difference was more like 6% slower. Furthermore, the JIT makes the two effectively indistinguishable.

On devices without a JIT, caching field accesses is about 20% faster than repeatedly accessing the field. With a JIT, field access costs about the same as local access, so this isn't a worthwhile optimization unless you feel it makes your code easier to read. (This is true of final, static, and static final fields too.)

Always Measure

Before you start optimizing, make sure you have a problem that you need to solve. Make sure you can accurately measure your existing performance, or you won't be able to measure the benefit of the alternatives you try.

Every claim made in this document is backed up by a benchmark. The source to these benchmarks can be found in the code.google.com "dalvik" project.

The benchmarks are built with the Caliper microbenchmarking framework for Java. Microbenchmarks are hard to get right, so Caliper goes out of its way to do the hard work for you, and even detect some cases where you're not measuring what you think you're measuring (because, say, the VM has managed to optimize all your code away). We highly recommend you use Caliper to run your own microbenchmarks.

You may also find Traceview useful for profiling, but it's important to realize that it currently disables the JIT, which may cause it to misattribute time to code that the JIT may be able to win back. It's especially important after making changes suggested by Traceview data to ensure that the resulting code actually runs faster when run without Traceview.

For more help profiling and debugging your apps, see the following documents:

- Profiling with Traceview and dmtracedump
- Analyzing UI Performance with Systrace

Next Get started Next class

Improving Layout Performance

Previous Next Get started

Dependencies and prerequisites

- Android 1.5 (API Level 3) or higher

You should also read

- XML Layouts

Video

DevBytes: Optimising Layouts with Hierarchy Viewer

Layouts are a key part of Android applications that directly affect the user experience. If implemented poorly, your layout can lead to a memory hungry application with slow UIs. The Android SDK includes tools to help you identify problems in your layout performance, which when combined the lessons here, you will be able to implement smooth scrolling interfaces with a minimum memory footprint.

Lessons

Optimizing Layout Hierarchies

In the same way a complex web page can slow down load time, your layout hierarchy if too complex can also cause performance problems. This lesson shows how you can use SDK tools to inspect your layout and discover performance bottlenecks.

Re-using Layouts with <include/>

If your application UI repeats certain layout constructs in multiple places, this lesson shows you how to create efficient, re-usable

layout constructs, then include them in the appropriate UI layouts.

Loading Views On Demand

Beyond simply including one layout component within another layout, you might want to make the included layout visible only when it's needed, sometime after the activity is running. This lesson shows how you can improve your layout's initialization performance by loading portions of your layout on demand.

Making ListView Scrolling Smooth

If you've built an instance of `ListView` that contains complex or data-heavy content in each list item, the scroll performance of the list might suffer. This lesson provides some tips about how you can make your scrolling performance more smooth.

Next Get started Next class

Optimizing Layout Hierarchies

Previous Next Get started

This lesson teaches you to

1. Inspect Your Layout
2. Revise Your Layout
3. Use Lint

You should also read

- XML Layouts
- Layout Resource

It is a common misconception that using the basic layout structures leads to the most efficient layouts. However, each widget and layout you add to your application requires initialization, layout, and drawing. For example, using nested instances of `LinearLayout` can lead to an excessively deep view hierarchy. Furthermore, nesting several instances of `LinearLayout` that use the `layout_weight` parameter can be especially expensive as each child needs to be measured twice. This is particularly important when the layout is inflated repeatedly, such as when used in a `ListView` or `GridView`.

In this lesson you'll learn to use Hierarchy Viewer and Layoutopt to examine and optimize your layout.

Inspect Your Layout

The Android SDK tools include a tool called Hierarchy Viewer that allows you to analyze your layout while your application is running. Using this tool helps you discover bottlenecks in the layout performance.

Hierarchy Viewer works by allowing you to select running processes on a connected device or emulator, then display the layout tree. The traffic lights on each block represent its Measure, Layout and Draw performance, helping you identify potential issues.

For example, figure 1 shows a layout that's used as an item in a `ListView`. This layout shows a small bitmap image on the left and two stacked items of text on the right. It is especially important that layouts that will be inflated multiple times—such as this one—are optimized as the performance benefits will be multiplied.

Figure 1. Conceptual layout for an item in a `ListView`.

The `hierarchyviewer` tool is available in `<sdk>/tools/`. When opened, the Hierarchy Viewer shows a list of available devices and its running components. Click **Load View Hierarchy** to view the layout hierarchy of the selected component. For example, figure 2 shows the layout for the list item illustrated by figure 1.

Figure 2. Layout hierarchy for the layout in figure 1, using nested instances of `LinearLayout`.

Figure 3. Clicking a hierarchy node shows its

performance
times.

In figure 2, you can see there is a 3-level hierarchy with some problems laying out the text items. Clicking on the items shows the time taken for each stage of the process (figure 3). It becomes clear which items are taking the longest to measure, layout, and render, and where you should spend time optimizing.

The timings for rendering a complete list item using this layout are:

- Measure: 0.977ms
- Layout: 0.167ms
- Draw: 2.717ms

Revise Your Layout

Because the layout performance above slows down due to a nested `LinearLayout`, the performance might improve by flattening the layout—make the layout shallow and wide, rather than narrow and deep. A `RelativeLayout` as the root node allows for such layouts. So, when this design is converted to use `RelativeLayout`, you can see that the layout becomes a 2-level hierarchy. Inspection of the new layout looks like this:

Figure 4. Layout hierarchy for the layout in figure 1, using `RelativeLayout`.

Now rendering a list item takes:

- Measure: 0.598ms
- Layout: 0.110ms
- Draw: 2.146ms

Might seem like a small improvement, but this time is multiplied several times because this layout is used for every item in a list.

Most of this time difference is due to the use of `layout_weight` in the `LinearLayout` design, which can slow down the speed of measurement. It is just one example of how each layout has appropriate uses and you should carefully consider whether using layout weight is necessary.

Use Lint

It is always good practice to run the lint tool on your layout files to search for possible view hierarchy optimizations. Lint has replaced the Layoutopt tool and has much greater functionality. Some examples of lint rules are:

- Use compound drawables - A `LinearLayout` which contains an `ImageView` and a `TextView` can be more efficiently handled as a compound drawable.
- Merge root frame - If a `FrameLayout` is the root of a layout and does not provide background or padding etc, it can be replaced with a merge tag which is slightly more efficient.
- Useless leaf - A layout that has no children or no background can often be removed (since it is invisible) for a flatter and more efficient layout hierarchy.
- Useless parent - A layout with children that has no siblings, is not a `ScrollView` or a root layout, and does not have a background, can be removed and have its children moved directly into the parent for a flatter and more efficient layout hierarchy.
- Deep layouts - Layouts with too much nesting are bad for performance. Consider using flatter layouts such as `RelativeLayout` or `GridLayout` to improve performance. The default maximum depth is 10.

Another benefit of Lint is that it is integrated into Android Studio. Lint automatically runs whenever you compile your program. With Android Studio, you can also run lint inspections for a specific build variant, or for all build variants.

You can also manage inspection profiles and configure inspections within Android Studio with the **File>Settings>Project Settings** option. The Inspection Configuration page appears with the supported inspections.

Figure 5. Inspection Configuration

Lint has the ability to automatically fix some issues, provide suggestions for others and jump directly to the offending code for review.

Next Get started Next class

Re-using Layouts with <include/>

Previous Next Get started

This lesson teaches you to

1. Create a Re-usable Layout
2. Use the <include> Tag
3. Use the <merge> Tag

You should also read

- Layout Resources

Although Android offers a variety of widgets to provide small and re-usable interactive elements, you might also need to re-use larger components that require a special layout. To efficiently re-use complete layouts, you can use the `<include/>` and `<merge/>` tags to embed another layout inside the current layout.

Reusing layouts is particularly powerful as it allows you create reusable complex layouts. For example, a yes/no button panel, or custom progress bar with description text. It also means that any elements of your application that are common across multiple layouts can be extracted, managed separately, then included in each layout. So while you can create individual UI components by writing a custom `View`, you can do it even more easily by re-using a layout file.

Create a Re-usable Layout

If you already know the layout that you want to re-use, create a new

XML file and define the layout. For example, here's a layout from the G-Kenya codelab that defines a title bar to be included in each activity (`titlebar.xml`):

```
<FrameLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width=”match_parent”
    android:layout_height="wrap_content"
    android:background="@color/titlebar_bg">

    <ImageView android:layout_width="wrap_content"
               android:layout_height="wrap_content"
               android:src="@drawable/gafricalogo" />
</FrameLayout>
```

The root `View` should be exactly how you'd like it to appear in each layout to which you add this layout.

Use the <include> Tag

Inside the layout to which you want to add the re-usable component, add the `<include/>` tag. For example, here's a layout from the G-Kenya codelab that includes the title bar from above:

Here's the layout file:

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width=”match_parent”
    android:layout_height=”match_parent”
    android:background="@color/app_bg"
    android:gravity="center_horizontal">

    <include layout="@layout/titlebar"/>

    <TextView android:layout_width=”match_parent”
              android:layout_height="wrap_content"
              android:text="@string/hello"
              android:padding="10dp" />

    ...

</LinearLayout>
```

You can also override all the layout parameters (any `android:layout_*` attributes) of the included layout's root view by specifying them in the `<include/>` tag. For example:

```
<include android:id="@+id/news_title"
         android:layout_width="match_parent"
         android:layout_height="match_parent"
         layout="@layout/title"/>
```

However, if you want to override layout attributes using the `<include>` tag, you must override both `android:layout_height` and `android:layout_width` in order for other layout attributes to take effect.

Use the <merge> Tag

The `<merge />` tag helps eliminate redundant view groups in your view hierarchy when including one layout within another. For example, if your main layout is a vertical `LinearLayout` in which two consecutive views can be re-used in multiple layouts, then the re-usable layout in which you place the two views requires its own root view. However, using another `LinearLayout` as the root for the re-usable layout would result in a vertical `LinearLayout` inside a vertical `LinearLayout`. The nested `LinearLayout` serves no real purpose other than to slow down your UI performance.

To avoid including such a redundant view group, you can instead use the `<merge>` element as the root view for the re-usable layout. For example:

```
<merge xmlns:android="http://schemas.android.com/apk/res/android">

    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/add"/>

    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/delete"/>

</merge>
```

Now, when you include this layout in another layout (using the `<include/>` tag), the system ignores the `<merge>` element and places the two buttons directly in the layout, in place of the `<include/>` tag.

Next Get started Next class

Loading Views On Demand

Previous Next Get started

This lesson teaches you to

1. Define a ViewStub
2. Load the ViewStub Layout

You should also read

- Optimize with stubs (blog post)

Sometimes your layout might require complex views that are rarely used. Whether they are item details, progress indicators, or undo messages, you can reduce memory usage and speed up rendering by loading the views only when they are needed.

Define a ViewStub

`ViewStub` is a lightweight view with no dimension and doesn't draw anything or participate in the layout. As such, it's cheap to inflate and cheap to leave in a view hierarchy. Each `ViewStub` simply needs to include the `android:layout` attribute to specify the layout to inflate.

The following `ViewStub` is for a translucent progress bar overlay. It should be visible only when new items are being imported into the application.

```
<ViewStub
    android:id="@+id/stub_import"
    android:inflatedId="@+id/panel_import"
    android:layout="@layout/progress_overlay"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="bottom" />
```

Load the ViewStub Layout

When you want to load the layout specified by the `ViewStub`, either set it visible by calling `setVisibility(View.VISIBLE)` or call `inflate()`.

```
((ViewStub)
findViewById(R.id.stub_import)).setVisibility(View.VISIBLE);
// or
View importPanel = ((ViewStub)
findViewById(R.id.stub_import)).inflate();
```

Note: The `inflate()` method returns the inflated `View` once complete. so you don't need to call `findViewById()` if you need to interact with the layout.

Once visible/inflated, the `ViewStub` element is no longer part of the view hierarchy. It is replaced by the inflated layout and the ID for the root view of that layout is the one specified by the `android:inflatedId` attribute of the ViewStub. (The ID `android:id` specified for the `ViewStub` is valid only until the `ViewStub` layout is visible/inflated.)

Note: One drawback of `ViewStub` is that it doesn’t currently support the `<merge/>` tag in the layouts to be inflated.

Next Get started Next class

Making ListView Scrolling Smooth

Previous Next Get started

This lesson teaches you to

1. Use a Background Thread
2. Hold View Objects in a View Holder

You should also read

- Why is my list black? An Android optimization

The key to a smoothly scrolling `ListView` is to keep the application's main thread (the UI thread) free from heavy processing. Ensure you do any disk access, network access, or SQL access in a separate thread. To test the status of your app, you can enable `StrictMode`.

Use a Background Thread

Using a background thread ("worker thread") removes strain from the main thread so it can focus on drawing the UI. In many cases, using `AsyncTask` provides a simple way to perform your work outside the main thread. `AsyncTask` automatically queues up all the `execute()` requests and performs them serially. This behavior is global to a particular process and means you don't need to worry about creating your own thread pool.

In the sample code below, an `AsyncTask` is used to load images in a background thread, then apply them to the UI once finished. It also shows a progress spinner in place of the images while they are loading.

```
// Using an AsyncTask to load the slow images in a background thread
```

```
new AsyncTask<ViewHolder, Void, Bitmap>() {
    private ViewHolder v;

    @Override
    protected Bitmap doInBackground(ViewHolder... params) {
        v = params[0];
        return mFakeImageLoader.getImage();
    }

    @Override
    protected void onPostExecute(Bitmap result) {
        super.onPostExecute(result);
        if (v.position == position) {
            // If this item hasn't been recycled already, hide the
            // progress and set and show the image
            v.progress.setVisibility(View.GONE);
            v.icon.setVisibility(View.VISIBLE);
            v.icon.setImageBitmap(result);
        }
    }
}.execute(holder);
```

Beginning with Android 3.0 (API level 11), an extra feature is available in `AsyncTask` so you can enable it to run across multiple processor cores. Instead of calling `execute()` you can specify `executeOnExecutor()` and multiple requests can be executed at the same time depending on the number of cores available.

Hold View Objects in a View Holder

Your code might call `findViewById()` frequently during the scrolling of `ListView`, which can slow down performance. Even when the `Adapter` returns an inflated view for recycling, you still need to look up the elements and update them. A way around repeated use of `findViewById()` is to use the "view holder" design pattern.

A `ViewHolder` object stores each of the component views inside the tag field of the Layout, so you can immediately access them without the need to look them up repeatedly. First, you need to create a class to hold your exact set of views. For example:

```
static class ViewHolder {
  TextView text;
  TextView timestamp;
  ImageView icon;
  ProgressBar progress;
  int position;
}
```

Then populate the `ViewHolder` and store it inside the layout.

```
ViewHolder holder = new ViewHolder();
holder.icon = (ImageView)
convertView.findViewById(R.id.listitem_image);
holder.text = (TextView)
convertView.findViewById(R.id.listitem_text);
holder.timestamp = (TextView)
convertView.findViewById(R.id.listitem_timestamp);
holder.progress = (ProgressBar)
convertView.findViewById(R.id.progress_spinner);
convertView.setTag(holder);
```

Now you can easily access each view without the need for the look-up, saving valuable processor cycles.

Next Get started Next class

Optimizing Battery Life

Previous Next Get started

Dependencies and prerequisites

- Android 2.0 (API level 5) or higher
- Experience with Intents and Intent Filters

You should also read

- Services

For your app to be a good citizen, it should seek to limit its impact on the battery life of its host device. After this class you will be able to build apps that monitor modify their functionality and behavior based on the state of the host device.

By taking steps such as disabling background service updates when you lose connectivity, or reducing the rate of such updates when the battery level is low, you can ensure that the impact of your app on battery life is minimized, without compromising the user experience.

Lessons

Monitoring the Battery Level and Charging State
: Learn how to alter your app's update rate by determining, and monitoring, the current battery level and changes in charging state.

Determining and Monitoring the Docking State and Type
: Optimal refresh rates can vary based on how the host device is being used. Learn how to determine, and monitor, the docking state and type of dock being used to affect your app's behavior.

Determining and Monitoring the Connectivity Status

Without Internet connectivity you can't update your app from an online source. Learn how to check the connectivity status to alter your background update rate. You'll also learn to check for Wi-Fi or mobile connectivity before beginning high-bandwidth operations.

Manipulating Broadcast Receivers On Demand

Broadcast receivers that you've declared in the manifest can be toggled at runtime to disable those that aren't necessary due to the current device state. Learn to improve efficiency by toggling and cascading state change receivers and delay actions until the device is in a specific state.

Next Get started Next class

Monitoring the Battery Level and Charging State

Previous Next Get started

This lesson teaches you to

1. Determine the Current Charging State
2. Monitor Changes in Charging State
3. Determine the Current Battery Level
4. Monitor Significant Changes in Battery Level

You should also read

- Intents and Intent Filters

When you're altering the frequency of your background updates to reduce the effect of those updates on battery life, checking the current battery level and charging state is a good place to start.

The battery-life impact of performing application updates depends on the battery level and charging state of the device. The impact of performing updates while the device is charging over AC is negligible, so in most cases you can maximize your refresh rate whenever the device is connected to a wall charger. Conversely, if the device is discharging, reducing your update rate helps prolong the battery life.

Similarly, you can check the battery charge level, potentially reducing the frequency of—or even stopping—your updates when the battery charge is nearly exhausted.

Determine the Current Charging State

Start by determining the current charge status. The `BatteryManager` broadcasts all battery and charging details in a sticky `Intent` that includes the charging status.

Because it's a sticky intent, you don't need to register a `BroadcastReceiver`—by simply calling `registerReceiver` passing in `null` as the receiver as shown in the next snippet, the current battery status intent is returned. You could pass in an actual `BroadcastReceiver` object here, but we'll be handling updates in a later section so it's not necessary.

```
IntentFilter ifilter = new
IntentFilter(Intent.ACTION_BATTERY_CHANGED);
Intent batteryStatus = context.registerReceiver(null, ifilter);
```

You can extract both the current charging status and, if the device is being charged, whether it's charging via USB or AC charger:

```
// 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;
```

Typically you should maximize the rate of your background updates in the case where the device is connected to an AC charger, reduce the rate if the charge is over USB, and lower it further if the battery is discharging.

Monitor Changes in Charging State

The charging status can change as easily as a device can be plugged in, so it's important to monitor the charging state for changes and alter your refresh rate accordingly.

The `BatteryManager` broadcasts an action whenever the device is

connected or disconnected from power. It's important to to receive these events even while your app isn't running—particularly as these events should impact how often you start your app in order to initiate a background update—so you should register a `BroadcastReceiver` in your manifest to listen for both events by defining the `ACTION_POWER_CONNECTED` and `ACTION_POWER_DISCONNECTED` within an intent filter.

```
<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>
```

Within the associated `BroadcastReceiver` implementation, you can extract the current charging state and method as described in the previous step.

```
public class PowerConnectionReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        int status = intent.getIntExtra(BatteryManager.EXTRA_STATUS,
-1);
        boolean isCharging = status ==
BatteryManager.BATTERY_STATUS_CHARGING ||
                             status ==
BatteryManager.BATTERY_STATUS_FULL;

        int chargePlug =
intent.getIntExtra(BatteryManager.EXTRA_PLUGGED, -1);
        boolean usbCharge = chargePlug ==
BatteryManager.BATTERY_PLUGGED_USB;
        boolean acCharge = chargePlug ==
BatteryManager.BATTERY_PLUGGED_AC;
    }
}
```

Determine the Current Battery Level

In some cases it's also useful to determine the current battery level. You may choose to reduce the rate of your background updates if the battery charge is below a certain level.

You can find the current battery charge by extracting the current battery level and scale from the battery status intent as shown here:

```
int level = batteryStatus.getIntExtra(BatteryManager.EXTRA_LEVEL, -
```

```
1);
int scale = batteryStatus.getIntExtra(BatteryManager.EXTRA_SCALE, -
1);

float batteryPct = level / (float)scale;
```

Monitor Significant Changes in Battery Level

You can't easily continually monitor the battery state, but you don't need to.

Generally speaking, the impact of constantly monitoring the battery level has a greater impact on the battery than your app's normal behavior, so it's good practice to only monitor significant changes in battery level—specifically when the device enters or exits a low battery state.

The manifest snippet below is extracted from the intent filter element within a broadcast receiver. The receiver is triggered whenever the device battery becomes low or exits the low condition by listening for `ACTION_BATTERY_LOW` and `ACTION_BATTERY_OKAY`.

```
<receiver android:name=".BatteryLevelReceiver">
<intent-filter>
  <action android:name="android.intent.action.ACTION_BATTERY_LOW"/>
  <action android:name="android.intent.action.ACTION_BATTERY_OKAY"/>
  </intent-filter>
</receiver>
```

It is generally good practice to disable all your background updates when the battery is critically low. It doesn't matter how fresh your data is if the phone turns itself off before you can make use of it.

In many cases, the act of charging a device is coincident with putting it into a dock. The next lesson shows you how to determine the current dock state and monitor for changes in device docking.

Next Get started Next class

Determining and Monitoring the Docking State and Type

Previous Next Get started

This lesson teaches you to

1. Determine the Current Docking State
2. Determine the Current Dock Type
3. Monitor for Changes in the Dock State or Type

You should also read

- Intents and Intent Filters

Android devices can be docked into several different kinds of docks. These include car or home docks and digital versus analog docks. The dock-state is typically closely linked to the charging state as many docks provide power to docked devices.

How the dock-state of the phone affects your update rate depends on your app. You may choose to increase the update frequency of a sports center app when it's in the desktop dock, or disable your updates completely if the device is car docked. Conversely, you may choose to maximize your updates while car docked if your background service is updating traffic conditions.

The dock state is also broadcast as a sticky `Intent`, allowing you to query if the device is docked or not, and if so, in which kind of dock.

Determine the Current Docking State

The dock-state details are included as an extra in a sticky broadcast of the `ACTION_DOCK_EVENT` action. Because it's sticky, you don't need to register a `BroadcastReceiver`. You can simply call `registerReceiver()` passing in `null` as the broadcast receiver as shown in the next snippet.

```
IntentFilter ifilter = new IntentFilter(Intent.ACTION_DOCK_EVENT);
Intent dockStatus = context.registerReceiver(null, ifilter);
```

You can extract the current docking status from the `EXTRA_DOCK_STATE` extra:

```
int dockState = battery.getIntExtra(EXTRA_DOCK_STATE, -1);
boolean isDocked = dockState != Intent.EXTRA_DOCK_STATE_UNDOCKED;
```

Determine the Current Dock Type

If a device is docked, it can be docked in any one of four different type of dock:

- Car
- Desk
- Low-End (Analog) Desk
- High-End (Digital) Desk

Note that the latter two options were only introduced to Android in API level 11, so it's good practice to check for all three where you are only interested in the type of dock rather than it being digital or analog specifically:

```
boolean isCar = dockState == EXTRA_DOCK_STATE_CAR;
boolean isDesk = dockState == EXTRA_DOCK_STATE_DESK ||
                 dockState == EXTRA_DOCK_STATE_LE_DESK ||
                 dockState == EXTRA_DOCK_STATE_HE_DESK;
```

Monitor for Changes in the Dock State or Type

Whenever the device is docked or undocked, the `ACTION_DOCK_EVENT` action is broadcast. To monitor changes in the device's dock-state, simply register a broadcast receiver in your application manifest as shown in the

snippet below:

```
<action android:name="android.intent.action.ACTION_DOCK_EVENT"/>
```

You can extract the dock type and state within the receiver implementation using the same techniques described in the previous step.

Next Get started Next class

Determining and Monitoring the Connectivity Status

Previous Next Get started

This lesson teaches you to

1. Determine if you Have an Internet Connection
2. Determine the Type of your Internet Connection
3. Monitor for Changes in Connectivity

You should also read

- Intents and Intent Filters

Some of the most common uses for repeating alarms and background services is to schedule regular updates of application data from Internet resources, cache data, or execute long running downloads. But if you aren't connected to the Internet, or the connection is too slow to complete your download, why both waking the device to schedule the update at all?

You can use the `ConnectivityManager` to check that you're actually connected to the Internet, and if so, what type of connection is in place.

Determine if You Have an Internet Connection

There's no need to schedule an update based on an Internet resource if you aren't connected to the Internet. The following snippet shows how to use the `ConnectivityManager` to query the active network and determine if it has Internet connectivity.

```
ConnectivityManager cm =

(ConnectivityManager)context.getSystemService(Context.CONNECTIVITY_SERV


NetworkInfo activeNetwork = cm.getActiveNetworkInfo();
boolean isConnected = activeNetwork != null &&
                      activeNetwork.isConnectedOrConnecting();
```

Determine the Type of your Internet Connection

It's also possible to determine the type of Internet connection currently available.

Device connectivity can be provided by mobile data, WiMAX, Wi-Fi, and ethernet connections. By querying the type of the active network, as shown below, you can alter your refresh rate based on the bandwidth available.

```
boolean isWiFi = activeNetwork.getType() ==
ConnectivityManager.TYPE_WIFI;
```

Mobile data costs tend to be significantly higher than Wi-Fi, so in most cases, your app's update rate should be lower when on mobile connections. Similarly, downloads of significant size should be suspended until you have a Wi-Fi connection.

Having disabled your updates, it's important that you listen for changes in connectivity in order to resume them once an Internet connection has been established.

Monitor for Changes in Connectivity

The `ConnectivityManager` broadcasts the `CONNECTIVITY_ACTION` (`"android.net.conn.CONNECTIVITY_CHANGE"`) action whenever the connectivity details have changed. You can register a broadcast receiver in your manifest to listen for these changes and resume (or suspend) your background updates accordingly.

```
<action android:name="android.net.conn.CONNECTIVITY_CHANGE"/>
```

Changes to a device's connectivity can be very frequent—this broadcast is triggered every time you move between mobile data and Wi-Fi. As a result, it's good practice to monitor this broadcast only when you've previously suspended updates or downloads in order to resume them. It's generally sufficient to simply check for Internet connectivity before beginning an update and, should there be none, suspend further updates until connectivity is restored.

This technique requires toggling broadcast receivers you've declard in the manifest, which is described in the next lesson.

Next Get started Next class

Manipulating Broadcast Receivers On Demand

Previous Next Get started

This lesson teaches you to

1. Toggle and Cascade State Change Receivers to Improve Efficiency

You should also read

- Intents and Intent Filters

The simplest way to monitor device state changes is to create a `BroadcastReceiver` for each state you're monitoring and register each of them in your application manifest. Then within each of these receivers you simply reschedule your recurring alarms based on the current device state.

A side-effect of this approach is that your app will wake the device each time any of these receivers is triggered—potentially much more frequently than required.

A better approach is to disable or enable the broadcast receivers at runtime. That way you can use the receivers you declared in the manifest as passive alarms that are triggered by system events only when necessary.

Toggle and Cascade State Change Receivers to Improve Efficiency

Use can use the `PackageManager` to toggle the enabled state on any

component defined in the manifest, including whichever broadcast receivers you wish to enable or disable as shown in the snippet below:

```
ComponentName receiver = new ComponentName(context,
myReceiver.class);

PackageManager pm = context.getPackageManager();

pm.setComponentEnabledSetting(receiver,
        PackageManager.COMPONENT_ENABLED_STATE_ENABLED,
        PackageManager.DONT_KILL_APP)
```

Using this technique, if you determine that connectivity has been lost, you can disable all of your receivers except the connectivity-change receiver. Conversely, once you are connected you can stop listening for connectivity changes and simply check to see if you're online immediately before performing an update and rescheduling a recurring update alarm.

You can use the same technique to delay a download that requires higher bandwidth to complete. Simply enable a broadcast receiver that listens for connectivity changes and initiates the download only after you are connected to Wi-Fi.

Next Get started Next class

Sending Operations to Multiple Threads

Previous Next Get started

Dependencies and prerequisites

- Android 3.0 (API Level 11) or higher
- Loading Data in the Background training class
- Running in a Background Service training class

You should also read

- Processes and Threads

Try it out

Download the sample

ThreadSample.zip

The speed and efficiency of a long-running, data-intensive operation often improves when you split it into smaller operations running on multiple threads. On a device that has a CPU with multiple processors (cores), the system can run the threads in parallel, rather than making each sub-operation wait for a chance to run. For example, decoding multiple image files in order to display them on a thumbnail screen runs substantially faster when you do each decode on a separate thread.

This class shows you how to set up and use multiple threads in an Android app, using a thread pool object. You'll also learn how to define code to run on a thread and how to communicate between one of these

threads and the UI thread.

Lessons

Specifying the Code to Run on a Thread

Learn how to write code to run on a separate `Thread`, by defining a class that implements the `Runnable` interface.

Creating a Manager for Multiple Threads

Learn how to create an object that manages a pool of `Thread` objects and a queue of `Runnable` objects. This object is called a `ThreadPoolExecutor`.

Running Code on a Thread Pool Thread

Learn how to run a `Runnable` on a thread from the thread pool.

Communicating with the UI Thread

Learn how to communicate from a thread in the thread pool to the UI thread.

Next Get started Next class

Specifying the Code to Run on a Thread

Previous Next Get started

This lesson teaches you to

1. Define a Class that Implements Runnable
2. Implement the run() Method

You should also read

- Processes and Threads

Try it out

Download the sample

ThreadSample.zip

This lesson shows you how to implement a `Runnable` class, which runs the code in its `Runnable.run()` method on a separate thread. You can also pass a `Runnable` to another object that can then attach it to a thread and run it. One or more `Runnable` objects that perform a particular operation are sometimes called a *task*.

`Thread` and `Runnable` are basic classes that, on their own, have only limited power. Instead, they're the basis of powerful Android classes such as `HandlerThread`, `AsyncTask`, and `IntentService`. `Thread` and `Runnable` are also the basis of the class `ThreadPoolExecutor`. This class automatically manages threads and task queues, and can even run multiple threads in parallel.

Define a Class that Implements Runnable

Implementing a class that implements `Runnable` is straightforward. For example:

```
public class PhotoDecodeRunnable implements Runnable {
    ...
    @Override
    public void run() {
        /*
         * Code you want to run on the thread goes here
         */
        ...
    }
    ...
}
```

Implement the run() Method

In the class, the `Runnable.run()` method contains the code that's executed. Usually, anything is allowable in a `Runnable`. Remember, though, that the `Runnable` won't be running on the UI thread, so it can't directly modify UI objects such as `View` objects. To communicate with the UI thread, you have to use the techniques described in the lesson Communicate with the UI Thread.

At the beginning of the `run()` method, set the thread to use background priority by calling `Process.setThreadPriority()` with `THREAD_PRIORITY_BACKGROUND`. This approach reduces resource competition between the `Runnable` object's thread and the UI thread.

You should also store a reference to the `Runnable` object's `Thread` in the `Runnable` itself, by calling `Thread.currentThread()`.

The following snippet shows how to set up the `run()` method:

```
class PhotoDecodeRunnable implements Runnable {
...
    /*
     * Defines the code to run for this task.
     */
    @Override
    public void run() {
        // Moves the current Thread into the background
```

```
android.os.Process.setThreadPriority(android.os.Process.THREAD_PRIORITY

        ...
        /*
         * Stores the current Thread in the PhotoTask instance,
         * so that the instance
         * can interrupt the Thread.
         */
        mPhotoTask.setImageDecodeThread(Thread.currentThread());
        ...
    }
...
}
```

Next Get started Next class

Creating a Manager for Multiple Threads

Previous Next Get started

This lesson teaches you to

1. Define the Thread Pool Class
2. Determine the Thread Pool Parameters
3. Create a Pool of Threads

You should also read

- Processes and Threads

Try it out

Download the sample

ThreadSample.zip

The previous lesson showed how to define a task that executes on a separate thread. If you only want to run the task once, this may be all you need. If you want to run a task repeatedly on different sets of data, but you only need one execution running at a time, an `IntentService` suits your needs. To automatically run tasks as resources become available, or to allow multiple tasks to run at the same time (or both), you need to provide a managed collection of threads. To do this, use an instance of `ThreadPoolExecutor`, which runs a task from a queue when a thread in its pool becomes free. To run a task, all you have to do is add it to the queue.

A thread pool can run multiple parallel instances of a task, so you should ensure that your code is thread-safe. Enclose variables that can be accessed by more than one thread in a `synchronized` block. This approach will prevent one thread from reading the variable while another is writing to it. Typically, this situation arises with static variables, but it also occurs in any object that is only instantiated once. To learn more about this, read the Processes and Threads API guide.

Define the Thread Pool Class

Instantiate `ThreadPoolExecutor` in its own class. Within this class, do the following:

Use static variables for thread pools

You may only want a single instance of a thread pool for your app, in order to have a single control point for restricted CPU or network resources. If you have different `Runnable` types, you may want to have a thread pool for each one, but each of these can be a single instance. For example, you can add this as part of your global field declarations:

```
public class PhotoManager {
    ...
    static  {
        ...
        // Creates a single static instance of PhotoManager
        sInstance = new PhotoManager();
    }
    ...
```

Use a private constructor

Making the constructor private ensures that it is a singleton, which means that you don't have to enclose accesses to the class in a `synchronized` block:

```
public class PhotoManager {
    ...
    /**
     * Constructs the work queues and thread pools used to
download
     * and decode images. Because the constructor is marked
private,
     * it's unavailable to other classes, even in the same
package.
     */
```

```
    private PhotoManager() {
    ...
    }
```

Start your tasks by calling methods in the thread pool class.

Define a method in the thread pool class that adds a task to a thread pool's queue. For example:

```
public class PhotoManager {
    ...
    // Called by the PhotoView to get a photo
    static public PhotoTask startDownload(
        PhotoView imageView,
        boolean cacheFlag) {
        ...
        // Adds a download task to the thread pool for
execution
        sInstance.
                mDownloadThreadPool.
                execute(downloadTask.getHTTPDownloadRunnable());
        ...
    }
```

Instantiate a `Handler` in the constructor and attach it to your app's UI thread.

A `Handler` allows your app to safely call the methods of UI objects such as `View` objects. Most UI objects may only be safely altered from the UI thread. This approach is described in more detail in the lesson Communicate with the UI Thread. For example:

```
    private PhotoManager() {
    ...
        // Defines a Handler object that's attached to the UI
thread
        mHandler = new Handler(Looper.getMainLooper()) {
            /*
             * handleMessage() defines the operations to perform
when
             * the Handler receives a new Message to process.
             */
            @Override
            public void handleMessage(Message inputMessage) {
                ...
            }
        ...
        }
    }
```

Determine the Thread Pool Parameters

Once you have the overall class structure, you can start defining the

thread pool. To instantiate a `ThreadPoolExecutor` object, you need the following values:

Initial pool size and maximum pool size
: The initial number of threads to allocate to the pool, and the maximum allowable number. The number of threads you can have in a thread pool depends primarily on the number of cores available for your device. This number is available from the system environment:

```
public class PhotoManager {
...
    /*
     * Gets the number of available cores
     * (not always the same as the maximum number of cores)
     */
    private static int NUMBER_OF_CORES =
            Runtime.getRuntime().availableProcessors();
}
```

This number may not reflect the number of physical cores in the device; some devices have CPUs that deactivate one or more cores depending on the system load. For these devices, `availableProcessors()` returns the number of *active* cores, which may be less than the total number of cores.

Keep alive time and time unit
: The duration that a thread will remain idle before it shuts down. The duration is interpreted by the time unit value, one of the constants defined in `TimeUnit`.

A queue of tasks
: The incoming queue from which `ThreadPoolExecutor` takes `Runnable` objects. To start code on a thread, a thread pool manager takes a `Runnable` object from a first-in, first-out queue and attaches it to the thread. You provide this queue object when you create the thread pool, using any queue class that implements the `BlockingQueue` interface. To match the requirements of your app, you can choose from the available queue implementations; to learn more about them, see the class overview for `ThreadPoolExecutor`. This example uses the `LinkedBlockingQueue` class:

```
public class PhotoManager {
    ...
    private PhotoManager() {
        ...
        // A queue of Runnables
```

```
            private final BlockingQueue<Runnable> mDecodeWorkQueue;
            ...
            // Instantiates the queue of Runnables as a
    LinkedBlockingQueue
            mDecodeWorkQueue = new LinkedBlockingQueue<Runnable>();
            ...
        }
        ...
    }
```

Create a Pool of Threads

To create a pool of threads, instantiate a thread pool manager by calling `ThreadPoolExecutor()`. This creates and manages a constrained group of threads. Because the initial pool size and the maximum pool size are the same, `ThreadPoolExecutor` creates all of the thread objects when it is instantiated. For example:

```
    private PhotoManager() {
        ...
        // Sets the amount of time an idle thread waits before
terminating
        private static final int KEEP_ALIVE_TIME = 1;
        // Sets the Time Unit to seconds
        private static final TimeUnit KEEP_ALIVE_TIME_UNIT =
TimeUnit.SECONDS;
        // Creates a thread pool manager
        mDecodeThreadPool = new ThreadPoolExecutor(
                NUMBER_OF_CORES,       // Initial pool size
                NUMBER_OF_CORES,       // Max pool size
                KEEP_ALIVE_TIME,
                KEEP_ALIVE_TIME_UNIT,
                mDecodeWorkQueue);
    }
```

Next Get started Next class

Running Code on a Thread Pool Thread

Previous Next Get started

This lesson teaches you to

1. Run a Runnable on a Thread in the Thread Pool
2. Interrupt Running Code

You should also read

- Processes and Threads

Try it out

Download the sample

ThreadSample.zip

The previous lesson showed you how to define a class that manages thread pools and the tasks that run on them. This lesson shows you how to run a task on a thread pool. To do this, you add the task to the pool's work queue. When a thread becomes available, the `ThreadPoolExecutor` takes a task from the queue and runs it on the thread.

This lesson also shows you how to stop a task that's running. You might want to do this if a task starts, but then discovers that its work isn't necessary. Rather than wasting processor time, you can cancel the thread the task is running on. For example, if you are downloading images from the network and using a cache, you probably want to stop a task if it detects that an image is already present in the cache. Depending

on how you write your app, you may not be able to detect this before you start the download.

Run a Task on a Thread in the Thread Pool

To start a task object on a thread in a particular thread pool, pass the `Runnable` to `ThreadPoolExecutor.execute()`. This call adds the task to the thread pool's work queue. When an idle thread becomes available, the manager takes the task that has been waiting the longest and runs it on the thread:

```
public class PhotoManager {
    public void handleState(PhotoTask photoTask, int state) {
        switch (state) {
            // The task finished downloading the image
            case DOWNLOAD_COMPLETE:
            // Decodes the image
                mDecodeThreadPool.execute(
                        photoTask.getPhotoDecodeRunnable());
            ...
        }
        ...
    }
    ...
}
```

When `ThreadPoolExecutor` starts a `Runnable` on a thread, it automatically calls the object's `run()` method.

Interrupt Running Code

To stop a task, you need to interrupt the task's thread. To prepare to do this, you need to store a handle to the task's thread when you create the task. For example:

```
class PhotoDecodeRunnable implements Runnable {
    // Defines the code to run for this task
    public void run() {
        /*
         * Stores the current Thread in the
         * object that contains PhotoDecodeRunnable
         */
        mPhotoTask.setImageDecodeThread(Thread.currentThread());
        ...
    }
    ...
}
```

To interrupt a thread, call `Thread.interrupt()`. Notice that `Thread` objects are controlled by the system, which can modify them outside of your app's process. For this reason, you need to lock access on a thread before you interrupt it, by placing the access in a `synchronized` block. For example:

```
public class PhotoManager {
    public static void cancelAll() {
        /*
         * Creates an array of Runnables that's the same size as the
         * thread pool work queue
         */
        Runnable[] runnableArray = new
Runnable[mDecodeWorkQueue.size()];
        // Populates the array with the Runnables in the queue
        mDecodeWorkQueue.toArray(runnableArray);
        // Stores the array length in order to iterate over the
array
        int len = runnableArray.length;
        /*
         * Iterates over the array of Runnables and interrupts each
one's Thread.
         */
        synchronized (sInstance) {
            // Iterates over the array of tasks
            for (int runnableIndex = 0; runnableIndex < len;
runnableIndex++) {
                // Gets the current thread
                Thread thread =
runnableArray[taskArrayIndex].mThread;
                // if the Thread exists, post an interrupt to it
                if (null != thread) {
                    thread.interrupt();
                }
            }
        }
    }
    ...
}
```

In most cases, `Thread.interrupt()` stops the thread immediately. However, it only stops threads that are waiting, and will not interrupt CPU or network-intensive tasks. To avoid slowing down or locking up the system, you should test for any pending interrupt requests before attempting an operation :

```
/*
 * Before continuing, checks to see that the Thread hasn't
 * been interrupted
 */
if (Thread.interrupted()) {
    return;
}
```

```
...
// Decodes a byte array into a Bitmap (CPU-intensive)
BitmapFactory.decodeByteArray(
        imageBuffer, 0, imageBuffer.length, bitmapOptions);
...
```

Next Get started Next class

Communicating with the UI Thread

Previous Next Get started

This lesson teaches you to

1. Define a Handler on the UI Thread
2. Move Data from a Task to the UI Thread

You should also read

- Processes and Threads

Try it out

Download the sample

ThreadSample.zip

In the previous lesson you learned how to start a task on a thread managed by `ThreadPoolExecutor`. This final lesson shows you how to send data from the task to objects running on the user interface (UI) thread. This feature allows your tasks to do background work and then move the results to UI elements such as bitmaps.

Every app has its own special thread that runs UI objects such as `View` objects; this thread is called the UI thread. Only objects running on the UI thread have access to other objects on that thread. Because tasks that you run on a thread from a thread pool *aren't* running on your UI thread, they don't have access to UI objects. To move data from a background thread to the UI thread, use a `Handler` that's running on the

UI thread.

Define a Handler on the UI Thread

Handler is part of the Android system's framework for managing threads. A Handler object receives messages and runs code to handle the messages. Normally, you create a Handler for a new thread, but you can also create a Handler that's connected to an existing thread. When you connect a Handler to your UI thread, the code that handles messages runs on the UI thread.

Instantiate the Handler object in the constructor for the class that creates your thread pools, and store the object in a global variable. Connect it to the UI thread by instantiating it with the Handler(Looper) constructor. This constructor uses a Looper object, which is another part of the Android system's thread management framework. When you instantiate a Handler based on a particular Looper instance, the Handler runs on the same thread as the Looper. For example:

```
private PhotoManager() {
...
    // Defines a Handler object that's attached to the UI thread
    mHandler = new Handler(Looper.getMainLooper()) {
    ...
```

Inside the Handler, override the handleMessage() method. The Android system invokes this method when it receives a new message for a thread it's managing; all of the Handler objects for a particular thread receive the same message. For example:

```
        /*
         * handleMessage() defines the operations to perform when
         * the Handler receives a new Message to process.
         */
        @Override
        public void handleMessage(Message inputMessage) {
            // Gets the image task from the incoming Message object.
            PhotoTask photoTask = (PhotoTask) inputMessage.obj;
            ...
        }
    ...
    }
}
```

The next section shows how to tell the Handler to move data.

Move Data from a Task to the UI Thread

To move data from a task object running on a background thread to an object on the UI thread, start by storing references to the data and the UI object in the task object. Next, pass the task object and a status code to the object that instantiated the `Handler`. In this object, send a `Message` containing the status and the task object to the `Handler`. Because `Handler` is running on the UI thread, it can move the data to the UI object.

Store data in the task object

For example, here's a `Runnable`, running on a background thread, that decodes a `Bitmap` and stores it in its parent object `PhotoTask`. The `Runnable` also stores the status code `DECODE_STATE_COMPLETED`.

```
// A class that decodes photo files into Bitmaps
class PhotoDecodeRunnable implements Runnable {
    ...
    PhotoDecodeRunnable(PhotoTask downloadTask) {
        mPhotoTask = downloadTask;
    }
    ...
    // Gets the downloaded byte array
    byte[] imageBuffer = mPhotoTask.getByteBuffer();
    ...
    // Runs the code for this task
    public void run() {
        ...
        // Tries to decode the image buffer
        returnBitmap = BitmapFactory.decodeByteArray(
                imageBuffer,
                0,
                imageBuffer.length,
                bitmapOptions
        );
        ...
        // Sets the ImageView Bitmap
        mPhotoTask.setImage(returnBitmap);
        // Reports a status of "completed"
        mPhotoTask.handleDecodeState(DECODE_STATE_COMPLETED);
        ...
    }
    ...
}
...
```

`PhotoTask` also contains a handle to the `ImageView` that displays the `Bitmap`. Even though references to the `Bitmap` and `ImageView` are in the same object, you can't assign the `Bitmap` to the `ImageView`, because

you're not currently running on the UI thread.

Instead, the next step is to send this status to the `PhotoTask` object.

Send status up the object hierarchy

`PhotoTask` is the next higher object in the hierarchy. It maintains references to the decoded data and the `View` object that will show the data. It receives a status code from `PhotoDecodeRunnable` and passes it along to the object that maintains thread pools and instantiates `Handler`:

```
public class PhotoTask {
    ...
    // Gets a handle to the object that creates the thread pools
    sPhotoManager = PhotoManager.getInstance();
    ...
    public void handleDecodeState(int state) {
        int outState;
        // Converts the decode state to the overall state.
        switch(state) {
            case PhotoDecodeRunnable.DECODE_STATE_COMPLETED:
                outState = PhotoManager.TASK_COMPLETE;
                break;
            ...
        }
        ...
        // Calls the generalized state method
        handleState(outState);
    }
    ...
    // Passes the state to PhotoManager
    void handleState(int state) {
        /*
         * Passes a handle to this task and the
         * current state to the class that created
         * the thread pools
         */
        sPhotoManager.handleState(this, state);
    }
    ...
}
```

Move data to the UI

From the `PhotoTask` object, the `PhotoManager` object receives a status code and a handle to the `PhotoTask` object. Because the status is `TASK_COMPLETE`, creates a `Message` containing the state and task object and sends it to the `Handler`:

```
public class PhotoManager {
```

```
    ...
    // Handle status messages from tasks
    public void handleState(PhotoTask photoTask, int state) {
        switch (state) {
            ...
            // The task finished downloading and decoding the image
            case TASK_COMPLETE:
                /*
                 * Creates a message for the Handler
                 * with the state and the task object
                 */
                Message completeMessage =
                        mHandler.obtainMessage(state, photoTask);
                completeMessage.sendToTarget();
                break;
            ...
        }
        ...
    }
```

Finally, `Handler.handleMessage()` checks the status code for each incoming `Message`. If the status code is `TASK_COMPLETE`, then the task is finished, and the `PhotoTask` object in the `Message` contains both a `Bitmap` and an `ImageView`. Because `Handler.handleMessage()` is running on the UI thread, it can safely move the `Bitmap` to the `ImageView`:

```
    private PhotoManager() {
        ...
            mHandler = new Handler(Looper.getMainLooper()) {
                @Override
                public void handleMessage(Message inputMessage) {
                    // Gets the task from the incoming Message
object.
                    PhotoTask photoTask = (PhotoTask)
inputMessage.obj;
                    // Gets the ImageView for this task
                    PhotoView localView = photoTask.getPhotoView();
                    ...
                    switch (inputMessage.what) {
                        ...
                        // The decoding is done
                        case TASK_COMPLETE:
                            /*
                             * Moves the Bitmap from the task
                             * to the View
                             */

localView.setImageBitmap(photoTask.getImage());
                            break;
                        ...
                        default:
                            /*
                             * Pass along other messages from the UI
                             */
                            super.handleMessage(inputMessage);
                    }
                    ...
```

```
                    }
                    ...
            }
            ...
    }
...
}
```

Next Get started Next class

Keeping Your App Responsive

In this document

1. What Triggers ANR?
2. How to Avoid ANRs
3. Reinforcing Responsiveness

Figure 1. An ANR dialog displayed to the user.

It's possible to write code that wins every performance test in the world, but still feels sluggish, hang or freeze for significant periods, or take too long to process input. The worst thing that can happen to your app's responsiveness is an "Application Not Responding" (ANR) dialog.

In Android, the system guards against applications that are insufficiently responsive for a period of time by displaying a dialog that says your app has stopped responding, such as the dialog in Figure 1. At this point, your app has been unresponsive for a considerable period of time so the system offers the user an option to quit the app. It's critical to design responsiveness into your application so the system never displays an ANR dialog to the user.

This document describes how the Android system determines whether an application is not responding and provides guidelines for ensuring that your application stays responsive.

What Triggers ANR?

Generally, the system displays an ANR if an application cannot respond to user input. For example, if an application blocks on some I/O operation (frequently a network access) on the UI thread so the system can't process incoming user input events. Or perhaps the app spends too much time building an elaborate in-memory structure or computing the next move in a game on the UI thread. It's always important to make sure these computations are efficient, but even the most efficient code still takes time to run.

In any situation in which your app performs a potentially lengthy operation, **you should not perform the work on the UI thread**, but instead create a worker thread and do most of the work there. This keeps the UI thread (which drives the user interface event loop) running and prevents the system from concluding that your code has frozen. Because such threading usually is accomplished at the class level, you can think of responsiveness as a *class* problem. (Compare this with basic code performance, which is a *method*-level concern.)

In Android, application responsiveness is monitored by the Activity Manager and Window Manager system services. Android will display the ANR dialog for a particular application when it detects one of the following conditions:

- No response to an input event (such as key press or screen touch events) within 5 seconds.
- A `BroadcastReceiver` hasn't finished executing within 10 seconds.

How to Avoid ANRs

Android applications normally run entirely on a single thread by default the "UI thread" or "main thread"). This means anything your application is doing in the UI thread that takes a long time to complete can trigger the ANR dialog because your application is not giving itself a chance to handle the input event or intent broadcasts.

Therefore, any method that runs in the UI thread should do as little work as possible on that thread. In particular, activities should do as little as possible to set up in key life-cycle methods such as onCreate() and onResume(). Potentially long running operations such as network or database operations, or computationally expensive calculations such as resizing bitmaps should be done in a worker thread (or in the case of databases operations, via an asynchronous request).

The most effecive way to create a worker thread for longer operations is with the AsyncTask class. Simply extend AsyncTask and implement the doInBackground() method to perform the work. To post progress changes to the user, you can call publishProgress(), which invokes the onProgressUpdate() callback method. From your implementation of onProgressUpdate() (which runs on the UI thread), you can notify the user. For example:

```
private class DownloadFilesTask extends AsyncTask<URL, Integer,
Long> {
    // Do the long-running work in here
    protected Long doInBackground(URL... urls) {
        int count = urls.length;
        long totalSize = 0;
        for (int i = 0; i < count; i++) {
            totalSize += Downloader.downloadFile(urls[i]);
            publishProgress((int) ((i / (float) count) * 100));
            // Escape early if cancel() is called
            if (isCancelled()) break;
        }
        return totalSize;
    }

    // This is called each time you call publishProgress()
    protected void onProgressUpdate(Integer... progress) {
        setProgressPercent(progress[0]);
    }

    // This is called when doInBackground() is finished
    protected void onPostExecute(Long result) {
        showNotification("Downloaded " + result + " bytes");
    }
}
```

To execute this worker thread, simply create an instance and call execute():

```
new DownloadFilesTask().execute(url1, url2, url3);
```

Although it's more complicated than AsyncTask, you might want to

instead create your own `Thread` or `HandlerThread` class. If you do, you should set the thread priority to "background" priority by calling `Process.setThreadPriority()` and passing `THREAD_PRIORITY_BACKGROUND`. If you don't set the thread to a lower priority this way, then the thread could still slow down your app because it operates at the same priority as the UI thread by default.

If you implement `Thread` or `HandlerThread`, be sure that your UI thread does not block while waiting for the worker thread to complete—do not call `Thread.wait()` or `Thread.sleep()`. Instead of blocking while waiting for a worker thread to complete, your main thread should provide a `Handler` for the other threads to post back to upon completion. Designing your application in this way will allow your app's UI thread to remain responsive to input and thus avoid ANR dialogs caused by the 5 second input event timeout.

The specific constraint on `BroadcastReceiver` execution time emphasizes what broadcast receivers are meant to do: small, discrete amounts of work in the background such as saving a setting or registering a `Notification`. So as with other methods called in the UI thread, applications should avoid potentially long-running operations or calculations in a broadcast receiver. But instead of doing intensive tasks via worker threads, your application should start an `IntentService` if a potentially long running action needs to be taken in response to an intent broadcast.

Tip: You can use `StrictMode` to help find potentially long running operations such as network or database operations that you might accidentally be doing your main thread.

Reinforce Responsiveness

Generally, 100 to 200ms is the threshold beyond which users will perceive slowness in an application. As such, here are some additional tips beyond what you should do to avoid ANR and make your application seem responsive to users:

If your application is doing work in the background in response to user input, show that progress is being made (such as with a `ProgressBar` in your UI).
- For games specifically, do calculations for moves in a worker thread.
- If your application has a time-consuming initial setup phase, consider showing a splash screen or rendering the main view as quickly as possible, indicate that loading is in progress and fill the information asynchronously. In either case, you should indicate somehow that progress is being made, lest the user perceive that the application is frozen.
- Use performance tools such as Systrace and Traceview to determine bottlenecks in your app's responsiveness.

Next Get started Next class

JNI Tips

In this document

1. JavaVM and JNIEnv
2. Threads
3. jclass, jmethodID, and jfieldID
4. Local and Global References
5. UTF-8 and UTF-16 Strings
6. Primitive Arrays
7. Region Calls
8. Exceptions
9. Extended Checking
10. Native Libraries
11. 64-bit Considerations
12. Unsupported Features/Backwards Compatibility
13. FAQ: Why do I get `UnsatisfiedLinkError`
14. FAQ: Why didn't `FindClass` find my class?
15. FAQ: How do I share raw data with native code?

JNI is the Java Native Interface. It defines a way for managed code (written in the Java programming language) to interact with native code (written in C/C++). It's vendor-neutral, has support for loading code from dynamic shared libraries, and while cumbersome at times is reasonably efficient.

If you're not already familiar with it, read through the Java Native Interface Specification to get a sense for how JNI works and what features are available. Some aspects of the interface aren't immediately obvious on first reading, so you may find the next few sections handy.

JavaVM and JNIEnv

JNI defines two key data structures, "JavaVM" and "JNIEnv". Both of these are essentially pointers to pointers to function tables. (In the C++ version, they're classes with a pointer to a function table and a member function for each JNI function that indirects through the table.) The JavaVM provides the "invocation interface" functions, which allow you to create and destroy a JavaVM. In theory you can have multiple JavaVMs per process, but Android only allows one.

The JNIEnv provides most of the JNI functions. Your native functions all receive a JNIEnv as the first argument.

The JNIEnv is used for thread-local storage. For this reason, **you cannot share a JNIEnv between threads**. If a piece of code has no other way to get its JNIEnv, you should share the JavaVM, and use `GetEnv` to discover the thread's JNIEnv. (Assuming it has one; see `AttachCurrentThread` below.)

The C declarations of JNIEnv and JavaVM are different from the C++ declarations. The `"jni.h"` include file provides different typedefs depending on whether it's included into C or C++. For this reason it's a bad idea to include JNIEnv arguments in header files included by both languages. (Put another way: if your header file requires `#ifdef __cplusplus`, you may have to do some extra work if anything in that header refers to JNIEnv.)

Threads

All threads are Linux threads, scheduled by the kernel. They're usually started from managed code (using `Thread.start`), but they can also be created elsewhere and then attached to the JavaVM. For example, a thread started with `pthread_create` can be attached with the JNI `AttachCurrentThread` or `AttachCurrentThreadAsDaemon` functions. Until a thread is attached, it has no JNIEnv, and **cannot make JNI calls**.

Attaching a natively-created thread causes a `java.lang.Thread` object to be constructed and added to the "main" `ThreadGroup`, making it visible to the debugger. Calling `AttachCurrentThread` on an already-attached thread is a no-op.

Android does not suspend threads executing native code. If garbage collection is in progress, or the debugger has issued a suspend request, Android will pause the thread the next time it makes a JNI call.

Threads attached through JNI **must call `DetachCurrentThread` before they exit**. If coding this directly is awkward, in Android 2.0 (Eclair) and higher you can use `pthread_key_create` to define a destructor function that will be called before the thread exits, and call `DetachCurrentThread` from there. (Use that key with `pthread_setspecific` to store the JNIEnv in thread-local-storage; that way it'll be passed into your destructor as the argument.)

jclass, jmethodID, and jfieldID

If you want to access an object's field from native code, you would do the following:

- Get the class object reference for the class with `FindClass`
- Get the field ID for the field with `GetFieldID`
- Get the contents of the field with something appropriate, such as `GetIntField`

Similarly, to call a method, you'd first get a class object reference and then a method ID. The IDs are often just pointers to internal runtime data structures. Looking them up may require several string comparisons, but once you have them the actual call to get the field or invoke the method is very quick.

If performance is important, it's useful to look the values up once and cache the results in your native code. Because there is a limit of one

JavaVM per process, it's reasonable to store this data in a static local structure.

The class references, field IDs, and method IDs are guaranteed valid until the class is unloaded. Classes are only unloaded if all classes associated with a ClassLoader can be garbage collected, which is rare but will not be impossible in Android. Note however that the `jclass` is a class reference and **must be protected** with a call to `NewGlobalRef` (see the next section).

If you would like to cache the IDs when a class is loaded, and automatically re-cache them if the class is ever unloaded and reloaded, the correct way to initialize the IDs is to add a piece of code that looks like this to the appropriate class:

```
    /*
     * We use a class initializer to allow the native code to cache
some
     * field offsets. This native function looks up and caches
interesting
     * class/field/method IDs. Throws on failure.
     */
    private static native void nativeInit();

    static {
        nativeInit();
    }
```

Create a `nativeClassInit` method in your C/C++ code that performs the ID lookups. The code will be executed once, when the class is initialized. If the class is ever unloaded and then reloaded, it will be executed again.

Local and Global References

Every argument passed to a native method, and almost every object returned by a JNI function is a "local reference". This means that it's valid for the duration of the current native method in the current thread. **Even if the object itself continues to live on after the native method returns, the reference is not valid.**

This applies to all sub-classes of `jobject`, including `jclass`, `jstring`, and `jarray`. (The runtime will warn you about most reference mis-uses when extended JNI checks are enabled.)

The only way to get non-local references is via the functions `NewGlobalRef` and `NewWeakGlobalRef`.

If you want to hold on to a reference for a longer period, you must use a "global" reference. The `NewGlobalRef` function takes the local reference as an argument and returns a global one. The global reference is guaranteed to be valid until you call `DeleteGlobalRef`.

This pattern is commonly used when caching a jclass returned from `FindClass`, e.g.:

```
jclass localClass = env->FindClass("MyClass");
jclass globalClass = reinterpret_cast<jclass>(env-
>NewGlobalRef(localClass));
```

All JNI methods accept both local and global references as arguments. It's possible for references to the same object to have different values. For example, the return values from consecutive calls to `NewGlobalRef` on the same object may be different. **To see if two references refer to the same object, you must use the `IsSameObject` function.** Never compare references with `==` in native code.

One consequence of this is that you **must not assume object references are constant or unique** in native code. The 32-bit value representing an object may be different from one invocation of a method to the next, and it's possible that two different objects could have the same 32-bit value on consecutive calls. Do not use `jobject` values as keys.

Programmers are required to "not excessively allocate" local references. In practical terms this means that if you're creating large numbers of local references, perhaps while running through an array of objects, you should free them manually with `DeleteLocalRef` instead of letting JNI do it for you. The implementation is only required to reserve slots for 16 local references, so if you need more than that you should either delete as you go or use `EnsureLocalCapacity`/`PushLocalFrame` to reserve more.

Note that `jfieldIDs` and `jmethodIDs` are opaque types, not object references, and should not be passed to `NewGlobalRef`. The raw data pointers returned by functions like `GetStringUTFChars` and `GetByteArrayElements` are also not objects. (They may be passed between threads, and are valid until the matching Release call.)

One unusual case deserves separate mention. If you attach a native thread with `AttachCurrentThread`, the code you are running will never automatically free local references until the thread detaches. Any local references you create will have to be deleted manually. In general, any native code that creates local references in a loop probably needs to do some manual deletion.

UTF-8 and UTF-16 Strings

The Java programming language uses UTF-16. For convenience, JNI provides methods that work with Modified UTF-8 as well. The modified encoding is useful for C code because it encodes \u0000 as 0xc0 0x80 instead of 0x00. The nice thing about this is that you can count on having C-style zero-terminated strings, suitable for use with standard libc string functions. The down side is that you cannot pass arbitrary UTF-8 data to JNI and expect it to work correctly.

If possible, it's usually faster to operate with UTF-16 strings. Android currently does not require a copy in `GetStringChars`, whereas `GetStringUTFChars` requires an allocation and a conversion to UTF-8. Note that **UTF-16 strings are not zero-terminated**, and \u0000 is allowed, so you need to hang on to the string length as well as the jchar pointer.

Don't forget to `Release` the strings you `Get`. The string functions return `jchar*` or `jbyte*`, which are C-style pointers to primitive data rather than local references. They are guaranteed valid until `Release` is called, which means they are not released when the native method returns.

Data passed to NewStringUTF must be in Modified UTF-8 format. A common mistake is reading character data from a file or network stream and handing it to `NewStringUTF` without filtering it. Unless you know the data is 7-bit ASCII, you need to strip out high-ASCII characters or convert them to proper Modified UTF-8 form. If you don't, the UTF-16 conversion will likely not be what you expect. The extended JNI checks will scan strings and warn you about invalid data, but they won't catch everything.

Primitive Arrays

JNI provides functions for accessing the contents of array objects. While arrays of objects must be accessed one entry at a time, arrays of primitives can be read and written directly as if they were declared in C.

To make the interface as efficient as possible without constraining the VM implementation, the `Get<PrimitiveType>ArrayElements` family of calls allows the runtime to either return a pointer to the actual elements, or allocate some memory and make a copy. Either way, the raw pointer returned is guaranteed to be valid until the corresponding `Release` call is issued (which implies that, if the data wasn't copied, the array object will be pinned down and can't be relocated as part of compacting the heap). **You must `Release` every array you `Get`.** Also, if the `Get` call fails, you must ensure that your code doesn't try to `Release` a NULL pointer later.

You can determine whether or not the data was copied by passing in a non-NULL pointer for the `isCopy` argument. This is rarely useful.

The `Release` call takes a `mode` argument that can have one of three values. The actions performed by the runtime depend upon whether it returned a pointer to the actual data or a copy of it:

- `0`
  - Actual: the array object is un-pinned.
  - Copy: data is copied back. The buffer with the copy is freed.
- `JNI_COMMIT`

  - Actual: does nothing.
  - Copy: data is copied back. The buffer with the copy **is not freed**.
- `JNI_ABORT`
  - Actual: the array object is un-pinned. Earlier writes are **not** aborted.
  - Copy: the buffer with the copy is freed; any changes to it are lost.

One reason for checking the `isCopy` flag is to know if you need to call `Release` with `JNI_COMMIT` after making changes to an array — if you're alternating between making changes and executing code that uses the contents of the array, you may be able to skip the no-op commit. Another possible reason for checking the flag is for efficient handling of `JNI_ABORT`. For example, you might want to get an array, modify it in place, pass pieces to other functions, and then discard the changes. If you know that JNI is making a new copy for you, there's no need to create another "editable" copy. If JNI is passing you the original, then you do need to make your own copy.

It is a common mistake (repeated in example code) to assume that you can skip the `Release` call if `*isCopy` is false. This is not the case. If no copy buffer was allocated, then the original memory must be pinned down and can't be moved by the garbage collector.

Also note that the `JNI_COMMIT` flag does **not** release the array, and you will need to call `Release` again with a different flag eventually.

Region Calls

There is an alternative to calls like `Get<Type>ArrayElements` and `GetStringChars` that may be very helpful when all you want to do is copy data in or out. Consider the following:

```
jbyte* data = env->GetByteArrayElements(array, NULL);
if (data != NULL) {
    memcpy(buffer, data, len);
```

```
        env->ReleaseByteArrayElements(array, data, JNI_ABORT);
    }
```

This grabs the array, copies the first `len` byte elements out of it, and then releases the array. Depending upon the implementation, the `Get` call will either pin or copy the array contents. The code copies the data (for perhaps a second time), then calls `Release`; in this case `JNI_ABORT` ensures there's no chance of a third copy.

One can accomplish the same thing more simply:

```
    env->GetByteArrayRegion(array, 0, len, buffer);
```

This has several advantages:

- Requires one JNI call instead of 2, reducing overhead.
- Doesn't require pinning or extra data copies.
- Reduces the risk of programmer error — no risk of forgetting to call `Release` after something fails.

Similarly, you can use the `Set<Type>ArrayRegion` call to copy data into an array, and `GetStringRegion` or `GetStringUTFRegion` to copy characters out of a `String`.

Exceptions

You must not call most JNI functions while an exception is pending. Your code is expected to notice the exception (via the function's return value, `ExceptionCheck`, or `ExceptionOccurred`) and return, or clear the exception and handle it.

The only JNI functions that you are allowed to call while an exception is pending are:

- `DeleteGlobalRef`
- `DeleteLocalRef`
- `DeleteWeakGlobalRef`
- `ExceptionCheck`
- `ExceptionClear`

ExceptionDescribe
- ExceptionOccurred
- MonitorExit
- PopLocalFrame
- PushLocalFrame
- Release<PrimitiveType>ArrayElements
- ReleasePrimitiveArrayCritical
- ReleaseStringChars
- ReleaseStringCritical
- ReleaseStringUTFChars

Many JNI calls can throw an exception, but often provide a simpler way of checking for failure. For example, if `NewString` returns a non-NULL value, you don't need to check for an exception. However, if you call a method (using a function like `CallObjectMethod`), you must always check for an exception, because the return value is not going to be valid if an exception was thrown.

Note that exceptions thrown by interpreted code do not unwind native stack frames, and Android does not yet support C++ exceptions. The JNI `Throw` and `ThrowNew` instructions just set an exception pointer in the current thread. Upon returning to managed from native code, the exception will be noted and handled appropriately.

Native code can "catch" an exception by calling `ExceptionCheck` or `ExceptionOccurred`, and clear it with `ExceptionClear`. As usual, discarding exceptions without handling them can lead to problems.

There are no built-in functions for manipulating the `Throwable` object itself, so if you want to (say) get the exception string you will need to find the `Throwable` class, look up the method ID for `getMessage "()Ljava/lang/String;"`, invoke it, and if the result is non-NULL use `GetStringUTFChars` to get something you can hand to `printf(3)` or equivalent.

Extended Checking

JNI does very little error checking. Errors usually result in a crash. Android also offers a mode called CheckJNI, where the JavaVM and JNIEnv function table pointers are switched to tables of functions that perform an extended series of checks before calling the standard implementation.

The additional checks include:

- Arrays: attempting to allocate a negative-sized array.
- Bad pointers: passing a bad jarray/jclass/jobject/jstring to a JNI call, or passing a NULL pointer to a JNI call with a non-nullable argument.
- Class names: passing anything but the “java/lang/String” style of class name to a JNI call.
- Critical calls: making a JNI call between a “critical” get and its corresponding release.
- Direct ByteBuffers: passing bad arguments to `NewDirectByteBuffer`.
- Exceptions: making a JNI call while there’s an exception pending.
- JNIEnv*s: using a JNIEnv* from the wrong thread.
- jfieldIDs: using a NULL jfieldID, or using a jfieldID to set a field to a value of the wrong type (trying to assign a StringBuilder to a String field, say), or using a jfieldID for a static field to set an instance field or vice versa, or using a jfieldID from one class with instances of another class.
- jmethodIDs: using the wrong kind of jmethodID when making a `Call*Method` JNI call: incorrect return type, static/non-static mismatch, wrong type for ‘this’ (for non-static calls) or wrong class (for static calls).
- References: using `DeleteGlobalRef`/`DeleteLocalRef` on the wrong kind of reference.
- Release modes: passing a bad release mode to a release call (something other than `0`, `JNI_ABORT`, or `JNI_COMMIT`).
- Type safety: returning an incompatible type from your native method (returning a StringBuilder from a method declared to return a String, say).
- UTF-8: passing an invalid Modified UTF-8 byte sequence to a JNI

call.

(Accessibility of methods and fields is still not checked: access restrictions don't apply to native code.)

There are several ways to enable CheckJNI.

If you’re using the emulator, CheckJNI is on by default.

If you have a rooted device, you can use the following sequence of commands to restart the runtime with CheckJNI enabled:

```
adb shell stop
adb shell setprop dalvik.vm.checkjni true
adb shell start
```

In either of these cases, you’ll see something like this in your logcat output when the runtime starts:

```
D AndroidRuntime: CheckJNI is ON
```

If you have a regular device, you can use the following command:

```
adb shell setprop debug.checkjni 1
```

This won’t affect already-running apps, but any app launched from that point on will have CheckJNI enabled. (Change the property to any other value or simply rebooting will disable CheckJNI again.) In this case, you’ll see something like this in your logcat output the next time an app starts:

```
D Late-enabling CheckJNI
```

You can also set the `android:debuggable` attribute in your application's manifest to turn on CheckJNI just for your app. Note that the Android build tools will do this automatically for certain build types.

Native Libraries

You can load native code from shared libraries with the standard `System.loadLibrary` call. The preferred way to get at your native code is:

- Call `System.loadLibrary` from a static class initializer. (See the earlier

example, where one is used to call `nativeClassInit`.) The argument is the "undecorated" library name, so to load "libfubar.so" you would pass in "fubar".

- Provide a native function: **`jint JNI_OnLoad(JavaVM* vm, void* reserved)`**
- In `JNI_OnLoad`, register all of your native methods. You should declare the methods "static" so the names don't take up space in the symbol table on the device.

The `JNI_OnLoad` function should look something like this if written in C++:

```
jint JNI_OnLoad(JavaVM* vm, void* reserved)
{
    JNIEnv* env;
    if (vm->GetEnv(reinterpret_cast<void**>(&env), JNI_VERSION_1_6)
!= JNI_OK) {
        return -1;
    }

    // Get jclass with env->FindClass.
    // Register methods with env->RegisterNatives.

    return JNI_VERSION_1_6;
}
```

You can also call `System.load` with the full path name of the shared library. For Android apps, you may find it useful to get the full path to the application's private data storage area from the context object.

This is the recommended approach, but not the only approach. Explicit registration is not required, nor is it necessary that you provide a `JNI_OnLoad` function. You can instead use "discovery" of native methods that are named in a specific way (see the JNI spec for details), though this is less desirable because if a method signature is wrong you won't know about it until the first time the method is actually used.

One other note about `JNI_OnLoad`: any `FindClass` calls you make from there will happen in the context of the class loader that was used to load the shared library. Normally `FindClass` uses the loader associated with the method at the top of the interpreted stack, or if there isn't one (because the thread was just attached) it uses the "system" class loader. This makes `JNI_OnLoad` a convenient place to look up and cache class object references.

64-bit Considerations

Android is currently expected to run on 32-bit platforms. In theory it could be built for a 64-bit system, but that is not a goal at this time. For the most part this isn't something that you will need to worry about when interacting with native code, but it becomes significant if you plan to store pointers to native structures in integer fields in an object. To support architectures that use 64-bit pointers, **you need to stash your native pointers in a `long` field rather than an `int`**.

Unsupported Features/Backwards Compatibility

All JNI 1.6 features are supported, with the following exception:

- `DefineClass` is not implemented. Android does not use Java bytecodes or class files, so passing in binary class data doesn't work.

For backward compatibility with older Android releases, you may need to be aware of:

- **Dynamic lookup of native functions**

  Until Android 2.0 (Eclair), the '$' character was not properly converted to "_00024" during searches for method names. Working around this requires using explicit registration or moving the native methods out of inner classes.

- **Detaching threads**

  Until Android 2.0 (Eclair), it was not possible to use a `pthread_key_create` destructor function to avoid the "thread must be detached before exit" check. (The runtime also uses a pthread key destructor function, so it'd be a race to see which gets called first.)

- **Weak global references**

  Until Android 2.2 (Froyo), weak global references were not implemented. Older versions will vigorously reject attempts to use them. You can use the Android platform version constants to test for support.

  Until Android 4.0 (Ice Cream Sandwich), weak global references could only be passed to `NewLocalRef`, `NewGlobalRef`, and `DeleteWeakGlobalRef`. (The spec strongly encourages programmers to create hard references to weak globals before doing anything with them, so this should not be at all limiting.)

  From Android 4.0 (Ice Cream Sandwich) on, weak global references can be used like any other JNI references.

- **Local references**

  Until Android 4.0 (Ice Cream Sandwich), local references were actually direct pointers. Ice Cream Sandwich added the indirection necessary to support better garbage collectors, but this means that lots of JNI bugs are undetectable on older releases. See JNI Local Reference Changes in ICS for more details.

- **Determining reference type with `GetObjectRefType`**

  Until Android 4.0 (Ice Cream Sandwich), as a consequence of the use of direct pointers (see above), it was impossible to implement `GetObjectRefType` correctly. Instead we used a heuristic that looked through the weak globals table, the arguments, the locals table, and the globals table in that order. The first time it found your direct pointer, it would report that your reference was of the type it happened to be examining. This meant, for example, that if you called `GetObjectRefType` on a global jclass that happened to be the same as the jclass passed as an implicit argument to your static native method, you'd get `JNILocalRefType` rather than `JNIGlobalRefType`.

FAQ: Why do I get `UnsatisfiedLinkError`?

When working on native code it's not uncommon to see a failure like this:

```
java.lang.UnsatisfiedLinkError: Library foo not found
```

In some cases it means what it says — the library wasn't found. In other cases the library exists but couldn't be opened by `dlopen(3)`, and the details of the failure can be found in the exception's detail message.

Common reasons why you might encounter "library not found" exceptions:

- The library doesn't exist or isn't accessible to the app. Use `adb shell ls -l <path>` to check its presence and permissions.
- The library wasn't built with the NDK. This can result in dependencies on functions or libraries that don't exist on the device.

Another class of `UnsatisfiedLinkError` failures looks like:

```
java.lang.UnsatisfiedLinkError: myfunc
        at Foo.myfunc(Native Method)
        at Foo.main(Foo.java:10)
```

In logcat, you'll see:

```
W/dalvikvm(  880): No implementation found for native LFoo;.myfunc
()V
```

This means that the runtime tried to find a matching method but was unsuccessful. Some common reasons for this are:

- The library isn't getting loaded. Check the logcat output for messages about library loading.
- The method isn't being found due to a name or signature mismatch. This is commonly caused by:
    - For lazy method lookup, failing to declare C++ functions with `extern "C"` and appropriate visibility (`JNIEXPORT`). Note that prior to Ice Cream Sandwich, the JNIEXPORT macro was incorrect, so

using a new GCC with an old `jni.h` won't work. You can use `arm-eabi-nm` to see the symbols as they appear in the library; if they look mangled (something like `_Z15Java_Foo_myfuncP7_JNIEnvP7_jclass` rather than `Java_Foo_myfunc`), or if the symbol type is a lowercase 't' rather than an uppercase 'T', then you need to adjust the declaration.
- For explicit registration, minor errors when entering the method signature. Make sure that what you're passing to the registration call matches the signature in the log file. Remember that 'B' is `byte` and 'Z' is `boolean`. Class name components in signatures start with 'L', end with ';', use '/' to separate package/class names, and use '$' to separate inner-class names (`Ljava/util/Map$Entry;`, say).

Using `javah` to automatically generate JNI headers may help avoid some problems.

FAQ: Why didn't `FindClass` find my class?

(Most of this advice applies equally well to failures to find methods with `GetMethodID` or `GetStaticMethodID`, or fields with `GetFieldID` or `GetStaticFieldID`.)

Make sure that the class name string has the correct format. JNI class names start with the package name and are separated with slashes, such as `java/lang/String`. If you're looking up an array class, you need to start with the appropriate number of square brackets and must also wrap the class with 'L' and ';', so a one-dimensional array of `String` would be `[Ljava/lang/String;`. If you're looking up an inner class, use '$' rather than '.'. In general, using `javap` on the .class file is a good way to find out the internal name of your class.

If you're using ProGuard, make sure that ProGuard didn't strip out your class. This can happen if your class/method/field is only used from JNI.

If the class name looks right, you could be running into a class loader issue. `FindClass` wants to start the class search in the class loader

associated with your code. It examines the call stack, which will look something like:

```
Foo.myfunc(Native Method)
Foo.main(Foo.java:10)
```

The topmost method is `Foo.myfunc`. `FindClass` finds the `ClassLoader` object associated with the `Foo` class and uses that.

This usually does what you want. You can get into trouble if you create a thread yourself (perhaps by calling `pthread_create` and then attaching it with `AttachCurrentThread`). Now there are no stack frames from your application. If you call `FindClass` from this thread, the JavaVM will start in the "system" class loader instead of the one associated with your application, so attempts to find app-specific classes will fail.

There are a few ways to work around this:

- Do your `FindClass` lookups once, in `JNI_OnLoad`, and cache the class references for later use. Any `FindClass` calls made as part of executing `JNI_OnLoad` will use the class loader associated with the function that called `System.loadLibrary` (this is a special rule, provided to make library initialization more convenient). If your app code is loading the library, `FindClass` will use the correct class loader.
- Pass an instance of the class into the functions that need it, by declaring your native method to take a Class argument and then passing `Foo.class` in.
- Cache a reference to the `ClassLoader` object somewhere handy, and issue `loadClass` calls directly. This requires some effort.

FAQ: How do I share raw data with native code?

You may find yourself in a situation where you need to access a large buffer of raw data from both managed and native code. Common

examples include manipulation of bitmaps or sound samples. There are two basic approaches.

You can store the data in a `byte[]`. This allows very fast access from managed code. On the native side, however, you're not guaranteed to be able to access the data without having to copy it. In some implementations, `GetByteArrayElements` and `GetPrimitiveArrayCritical` will return actual pointers to the raw data in the managed heap, but in others it will allocate a buffer on the native heap and copy the data over.

The alternative is to store the data in a direct byte buffer. These can be created with `java.nio.ByteBuffer.allocateDirect`, or the JNI `NewDirectByteBuffer` function. Unlike regular byte buffers, the storage is not allocated on the managed heap, and can always be accessed directly from native code (get the address with `GetDirectBufferAddress`). Depending on how direct byte buffer access is implemented, accessing the data from managed code can be very slow.

The choice of which to use depends on two factors:

1. Will most of the data accesses happen from code written in Java or in C/C++?
2. If the data is eventually being passed to a system API, what form must it be in? (For example, if the data is eventually passed to a function that takes a byte[], doing processing in a direct `ByteBuffer` might be unwise.)

If there's no clear winner, use a direct byte buffer. Support for them is built directly into JNI, and performance should improve in future releases.

Next Get started Next class

SMP Primer for Android

In this document

1. Theory
    1. Memory consistency models
        1. Processor consistency
        2. CPU cache behavior
        3. Observability
        4. ARM's weak ordering
    2. Data memory barriers
        1. Store/store and load/load
        2. Load/store and store/load
        3. Barrier instructions
        4. Address dependencies and causal consistency
        5. Memory barrier summary
    3. Atomic operations
        1. Atomic essentials
        2. Atomic + barrier pairing
        3. Acquire and release
2. Practice
    1. What not to do in C
        1. C/C++ and "volatile"

        2. Examples
    2. What not to do in Java
        1. “synchronized” and “volatile”
        2. Examples
    3. What to do
        1. General advice
        2. Synchronization primitive guarantees
        3. Upcoming changes to C/C++
3. Closing Notes
4. Appendix
    1. SMP failure example
    2. Implementing synchronization stores
    3. Further reading

Android 3.0 and later platform versions are optimized to support multiprocessor architectures. This document introduces issues that can arise when writing code for symmetric multiprocessor systems in C, C++, and the Java programming language (hereafter referred to simply as “Java” for the sake of brevity). It's intended as a primer for Android app developers, not as a complete discussion on the subject. The focus is on the ARM CPU architecture.

If you’re in a hurry, you can skip the Theory section and go directly to Practice for best practices, but this is not recommended.

Introduction

SMP is an acronym for “Symmetric Multi-Processor”. It describes a design in which two or more identical CPU cores share access to main memory. Until a few years ago, all Android devices were UP (Uni-Processor).

Most — if not all — Android devices do have multiple CPUs, but generally one of them is used to run applications while others manage various bits of device hardware (for example, the radio). The CPUs may have different architectures, and the programs running on them can't use main memory to communicate with each other.

Most Android devices sold today are built around SMP designs, making things a bit more complicated for software developers. The sorts of race conditions you might encounter in a multi-threaded program are much worse on SMP when two or more of your threads are running simultaneously on different cores. What's more, SMP on ARM is more challenging to work with than SMP on x86. Code that has been thoroughly tested on x86 may break badly on ARM.

The rest of this document will explain why, and tell you what you need to do to ensure that your code behaves correctly.

Theory

This is a high-speed, glossy overview of a complex subject. Some areas will be incomplete, but none of it should be misleading or wrong.

See Further reading at the end of the document for pointers to more thorough treatments of the subject.

Memory consistency models

Memory consistency models, or often just "memory models", describe the guarantees the hardware architecture makes about memory accesses. For example, if you write a value to address A, and then write a value to address B, the model might guarantee that every CPU core sees those writes happen in that order.

The model most programmers are accustomed to is *sequential consistency*, which is described like this (Adve &

Gharachorloo):

- All memory operations appear to execute one at a time
- All operations on a single processor appear to execute in the order described by that processor's program.

If you look at a bit of code and see that it does some reads and writes from memory, on a sequentially-consistent CPU architecture you know that the code will do those reads and writes in the expected order. It's possible that the CPU is actually reordering instructions and delaying reads and writes, but there is no way for code running on the device to tell that the CPU is doing anything other than execute instructions in a straightforward manner. (We're ignoring memory-mapped device driver I/O for the moment.)

To illustrate these points it's useful to consider small snippets of code, commonly referred to as *litmus tests*. These are assumed to execute in *program order*, that is, the order in which the instructions appear here is the order in which the CPU will execute them. We don't want to consider instruction reordering performed by compilers just yet.

Here's a simple example, with code running on two threads:

Thread 1	Thread 2
`A = 3`	`reg0 = B`
`B = 5`	`reg1 = A`

In this and all future litmus examples, memory locations are represented by capital letters (A, B, C) and CPU registers start with "reg". All memory is initially zero. Instructions are executed from top to bottom. Here, thread 1 stores the value 3 at location A, and then the value 5 at location B. Thread 2 loads the value from location B into reg0, and then loads the value from location A into reg1. (Note that we're writing in one order and reading in another.)

Thread 1 and thread 2 are assumed to execute on different CPU cores. You should **always** make this assumption when thinking about multi-threaded code.

Sequential consistency guarantees that, after both threads have finished executing, the registers will be in one of the following states:

Registers	States
reg0=5, reg1=3	possible (thread 1 ran first)
reg0=0, reg1=0	possible (thread 2 ran first)
reg0=0, reg1=3	possible (concurrent execution)
reg0=5, reg1=0	never

To get into a situation where we see B=5 before we see the store to A, either the reads or the writes would have to happen out of order. On a sequentially-consistent machine, that can't happen.

Most uni-processors, including x86 and ARM, are sequentially consistent. Most SMP systems, including x86 and ARM, are not.

Processor consistency

x86 SMP provides *processor consistency*, which is slightly weaker than sequential. While the architecture guarantees that loads are not reordered with respect to other loads, and stores are not reordered with respect to other stores, it does not guarantee that a store followed by a load will be observed in the expected order.

Consider the following example, which is a piece of Dekker's Algorithm for mutual exclusion:

Thread 1	Thread 2

```
A = true            B = true
reg1 = B            reg2 = A
if (reg1 == false) if (reg2 == false)
    critical-stuff     critical-stuff
```

The idea is that thread 1 uses A to indicate that it's busy, and thread 2 uses B. Thread 1 sets A and then checks to see if B is set; if not, it can safely assume that it has exclusive access to the critical section. Thread 2 does something similar. (If a thread discovers that both A and B are set, a turn-taking algorithm is used to ensure fairness.)

On a sequentially-consistent machine, this works correctly. On x86 and ARM SMP, the store to A and the load from B in thread 1 can be "observed" in a different order by thread 2. If that happened, we could actually appear to execute this sequence (where blank lines have been inserted to highlight the apparent order of operations):

```
    Thread 1            Thread 2
reg1 = B
                    B = true
                    reg2 = A
A = true
if (reg1 == false) if (reg2 == false)
    critical-stuff     critical-stuff
```

This results in both reg1 and reg2 set to "false", allowing the threads to execute code in the critical section simultaneously. To understand how this can happen, it's useful to know a little about CPU caches.

CPU cache behavior

This is a substantial topic in and of itself. An extremely brief overview follows. (The motivation for this material is to provide some basis for understanding why SMP systems behave as they do.)

Modern CPUs have one or more caches between the processor and main memory. These are labeled L1, L2, and so on, with the higher numbers being successively “farther” from the CPU. Cache memory adds size and cost to the hardware, and increases power consumption, so the ARM CPUs used in Android devices typically have small L1 caches and little or no L2/L3.

Loading or storing a value into the L1 cache is very fast. Doing the same to main memory can be 10-100x slower. The CPU will therefore try to operate out of the cache as much as possible. The *write policy* of a cache determines when data written to it is forwarded to main memory. A *write-through* cache will initiate a write to memory immediately, while a *write-back* cache will wait until it runs out of space and has to evict some entries. In either case, the CPU will continue executing instructions past the one that did the store, possibly executing dozens of them before the write is visible in main memory. (While the write-through cache has a policy of immediately forwarding the data to main memory, it only **initiates** the write. It does not have to wait for it to finish.)

The cache behavior becomes relevant to this discussion when each CPU core has its own private cache. In a simple model, the caches have no way to interact with each other directly. The values held by core #1’s cache are not shared with or visible to core #2’s cache except as loads or stores from main memory. The long latencies on memory accesses would make inter-thread interactions sluggish, so it’s useful to define a way for the caches to share data. This sharing is called *cache coherency*, and the coherency rules are defined by the CPU architecture’s *cache consistency model*.

With that in mind, let’s return to the Dekker example. When core 1 executes “A = 1”, the value gets stored in core 1’s cache. When core 2 executes “if (A == 0)”, it might read from main memory or it might read from core 2’s cache; either way it won’t see the store performed by core 1. (“A” could be in core 2’s cache because of a previous load from “A”.)

For the memory consistency model to be sequentially consistent, core 1 would have to wait for all other cores to be aware of “A = 1” before it could execute “if (B == 0)” (either through strict cache coherency rules, or by disabling the caches entirely so everything operates out of main memory). This would impose a performance penalty on every store operation. Relaxing the rules for the ordering of stores followed by loads improves performance but imposes a burden on software developers.

The other guarantees made by the processor consistency model are less expensive to make. For example, to ensure that memory writes are not observed out of order, it just needs to ensure that the stores are published to other cores in the same order that they were issued. It doesn’t need to wait for store #1 to **finish** being published before it can start on store #2, it just needs to ensure that it doesn’t finish publishing #2 before it finishes publishing #1. This avoids a performance bubble.

Relaxing the guarantees even further can provide additional opportunities for CPU optimization, but creates more opportunities for code to behave in ways the programmer didn’t expect.

One additional note: CPU caches don’t operate on individual bytes. Data is read or written as *cache lines*; for many ARM CPUs these are 32 bytes. If you read data from a location in main memory, you will also be reading some adjacent values. Writing data will cause the cache line to be read from memory and updated. As a result, you can cause a value to be loaded into cache as a side-effect of reading or writing something nearby, adding to the general aura of mystery.

Observability

Before going further, it’s useful to define in a more rigorous fashion what is meant by “observing” a load or store. Suppose core 1 executes “A = 1”. The store is *initiated* when the CPU executes the instruction. At some point later, possibly through cache coherence activity, the store is *observed* by core 2. In a write-through cache it doesn’t really *complete* until the store arrives in main memory, but the memory consistency model doesn’t dictate when something completes, just when it can be *observed*.

(In a kernel device driver that accesses memory-mapped I/O locations, it may be very important to know when things actually complete. We're not going to go into that here.)

Observability may be defined as follows:

- "A write to a location in memory is said to be observed by an observer Pn when a subsequent read of the location by Pn would return the value written by the write."
- "A read of a location in memory is said to be observed by an observer Pm when a subsequent write to the location by Pm would have no effect on the value returned by the read." (*Reasoning about the ARM weakly consistent memory model*)

A less formal way to describe it (where "you" and "I" are CPU cores) would be:

- I have observed your write when I can read what you wrote
- I have observed your read when I can no longer affect the value you read

The notion of observing a write is intuitive; observing a read is a bit less so (don't worry, it grows on you).

With this in mind, we're ready to talk about ARM.

ARM's weak ordering

ARM SMP provides weak memory consistency guarantees. It does not guarantee that loads or stores are ordered with respect to each other.

Thread 1	Thread 2
`A = 41`	`loop_until (B == 1)`
`B = 1 // "A is ready"`	`reg = A`

Recall that all addresses are initially zero. The “loop_until” instruction reads B repeatedly, looping until we read 1 from B. The idea here is that thread 2 is waiting for thread 1 to update A. Thread 1 sets A, and then sets B to 1 to indicate data availability.

On x86 SMP, this is guaranteed to work. Thread 2 will observe the stores made by thread 1 in program order, and thread 1 will observe thread 2’s loads in program order.

On ARM SMP, the loads and stores can be observed in any order. It is possible, after all the code has executed, for reg to hold 0. It’s also possible for it to hold 41. Unless you explicitly define the ordering, you don’t know how this will come out.

(For those with experience on other systems, ARM’s memory model is equivalent to PowerPC in most respects.)

Data memory barriers

Memory barriers provide a way for your code to tell the CPU that memory access ordering matters. ARM/x86 uniprocessors offer sequential consistency, and thus have no need for them. (The barrier instructions can be executed but aren’t useful; in at least one case they’re hideously expensive, motivating separate builds for SMP targets.)

There are four basic situations to consider:

1. store followed by another store
2. load followed by another load
3. load followed by store
4. store followed by load

Store/store and load/load

Recall our earlier example:

```
      Thread 1              Thread 2
A = 41                  loop_until (B == 1)
B = 1 // "A is ready"   reg = A
```

Thread 1 needs to ensure that the store to A happens before the store to B. This is a "store/store" situation. Similarly, thread 2 needs to ensure that the load of B happens before the load of A; this is a load/load situation. As mentioned earlier, the loads and stores can be observed in any order.

> *Going back to the cache discussion, assume A and B are on separate cache lines, with minimal cache coherency. If the store to A stays local but the store to B is published, core 2 will see B=1 but won't see the update to A. On the other side, assume we read A earlier, or it lives on the same cache line as something else we recently read. Core 2 spins until it sees the update to B, then loads A from its local cache, where the value is still zero.*

We can fix it like this:

```
      Thread 1              Thread 2
A = 41                  loop_until (B == 1)
store/store barrier     load/load barrier
B = 1 // "A is ready"   reg = A
```

The store/store barrier guarantees that **all observers** will observe the write to A before they observe the write to B. It makes no guarantees about the ordering of loads in thread 1, but we don't have any of those, so that's

okay. The load/load barrier in thread 2 makes a similar guarantee for the loads there.

Since the store/store barrier guarantees that thread 2 observes the stores in program order, why do we need the load/load barrier in thread 2? Because we also need to guarantee that thread 1 observes the loads in program order.

> *The store/store barrier could work by flushing all dirty entries out of the local cache, ensuring that other cores see them before they see any future stores. The load/load barrier could purge the local cache completely and wait for any "in-flight" loads to finish, ensuring that future loads are observed after previous loads. What the CPU actually does doesn't matter, so long as the appropriate guarantees are kept. If we use a barrier in core 1 but not in core 2, core 2 could still be reading A from its local cache.*

Because the architectures have different memory models, these barriers are required on ARM SMP but not x86 SMP.

Load/store and store/load

The Dekker's Algorithm fragment shown earlier illustrated the need for a store/load barrier. Here's an example where a load/store barrier is required:

Thread 1	Thread 2
`reg = A`	`loop_until (B == 1)`
`B = 1 // "I have latched A"`	`A = 41 // update A`

Thread 2 could observe thread 1's store of B=1 before it observe's thread 1's load from A, and as a result store

A=41 before thread 1 has a chance to read A. Inserting a load/store barrier in each thread solves the problem:

Thread 1	Thread 2
`reg = A`	`loop_until (B == 1)`
`load/store barrier`	*`load/store barrier`*
`B = 1 // “I have latched A”`	`A = 41 // update A`

> *A store to local cache may be observed before a load from main memory, because accesses to main memory are so much slower. In this case, assume core 1's cache has the cache line for B but not A. The load from A is initiated, and while that's in progress execution continues. The store to B happens in local cache, and by some means becomes available to core 2 while the load from A is still in progress. Thread 2 is able to exit the loop before it has observed thread 1's load from A.*
>
> *A thornier question is: do we need a barrier in thread 2? If the CPU doesn't perform speculative writes, and doesn't execute instructions out of order, can thread 2 store to A before thread 1's read if thread 1 guarantees the load/store ordering? (Answer: no.) What if there's a third core watching A and B? (Answer: now you need one, or you could observe B==0 / A==41 on the third core.) It's safest to insert barriers in both places and not worry about the details.*

As mentioned earlier, store/load barriers are the only kind required on x86 SMP.

Barrier instructions

Different CPUs provide different flavors of barrier instruction. For example:

- Sparc V8 has a “membar” instruction that takes a 4-element bit vector. The four categories of barrier can

be specified individually.

- Alpha provides “rmb” (load/load), “wmb” (store/store), and “mb” (full). (Trivia: the linux kernel provides three memory barrier functions with these names and behaviors.)
- x86 has a variety of options; “mfence” (introduced with SSE2) provides a full barrier.
- ARMv7 has “dmb st” (store/store) and “dmb sy” (full).

“Full barrier” means all four categories are included.

It is important to recognize that the only thing guaranteed by barrier instructions is ordering. Do not treat them as cache coherency “sync points” or synchronous “flush” instructions. The ARM “dmb” instruction has no direct effect on other cores. This is important to understand when trying to figure out where barrier instructions need to be issued.

Address dependencies and causal consistency

(This is a slightly more advanced topic and can be skipped.)

The ARM CPU provides one special case where a load/load barrier can be avoided. Consider the following example from earlier, modified slightly:

Thread 1	Thread 2
	`loop:`
`[A+8] = 41`	`reg0 = B`
`store/store barrier`	`if (reg0 == 0) goto loop`
`B = 1 // “A is ready”`	`reg1 = 8`
	`reg2 = [A + reg1]`

This introduces a new notation. If “A” refers to a memory address, “A+n” refers to a memory address offset by

8 bytes from A. If A is the base address of an object or array, [A+8] could be a field in the object or an element in the array.

The “loop_until” seen in previous examples has been expanded to show the load of B into reg0. reg1 is assigned the numeric value 8, and reg2 is loaded from the address [A+reg1] (the same location that thread 1 is accessing).

This will not behave correctly because the load from B could be observed after the load from [A+reg1]. We can fix this with a load/load barrier after the loop, but on ARM we can also just do this:

```
      Thread 1                 Thread 2

                           loop:
[A+8] = 41                     reg0 = B
store/store barrier            if (reg0 == 0) goto loop
B = 1 // “A is ready” reg1 = 8 + (reg0 & 0)
                           reg2 = [A + reg1]
```

What we’ve done here is change the assignment of reg1 from a constant (8) to a value that depends on what we loaded from B. In this case, we do a bitwise AND of the value with 0, which yields zero, which means reg1 still has the value 8. However, the ARM CPU believes that the load from [A+reg1] depends upon the load from B, and will ensure that the two are observed in program order.

This is called an *address dependency*. Address dependencies exist when the value returned by a load is used to compute the address of a subsequent load or store. It can let you avoid the need for an explicit barrier in certain situations.

ARM does not provide *control dependency* guarantees. To illustrate this it’s necessary to dip into ARM code for a moment: (*Barrier Litmus Tests and Cookbook*).

```
LDR r1, [r0]
CMP r1, #55
LDRNE r2, [r3]
```

The loads from r0 and r3 may be observed out of order, even though the load from r3 will not execute at all if [r0] doesn't hold 55. Inserting AND r1, r1, #0 and replacing the last instruction with LDRNE r2, [r3, r1] would ensure proper ordering without an explicit barrier. (This is a prime example of why you can't think about consistency issues in terms of instruction execution. Always think in terms of memory accesses.)

While we're hip-deep, it's worth noting that ARM does not provide *causal consistency*:

Thread 1	Thread 2	Thread 3

`A = 1`	`loop_until (A == 1)`
	`B = 1`
		`loop:`
		`  reg0 = B`
		`  if (reg0 == 0) goto loop`
		`reg1 = reg0 & 0`
		`reg2 = [A+reg1]`

Here, thread 1 sets A, signaling thread 2. Thread 2 sees that and sets B to signal thread 3. Thread 3 sees it and loads from A, using an address dependency to ensure that the load of B and the load of A are observed in program order.

It's possible for reg2 to hold zero at the end of this. The fact that a store in thread 1 causes something to happen in thread 2 which causes something to happen in thread 3 does not mean that thread 3 will observe the stores in that order. (Inserting a load/store barrier in thread 2 fixes this.)

Memory barrier summary

Barriers come in different flavors for different situations. While there can be performance advantages to using

exactly the right barrier type, there are code maintenance risks in doing so — unless the person updating the code fully understands it, they might introduce the wrong type of operation and cause a mysterious breakage. Because of this, and because ARM doesn't provide a wide variety of barrier choices, many atomic primitives use full barrier instructions when a barrier is required.

The key thing to remember about barriers is that they define ordering. Don't think of them as a "flush" call that causes a series of actions to happen. Instead, think of them as a dividing line in time for operations on the current CPU core.

Atomic operations

Atomic operations guarantee that an operation that requires a series of steps always behaves as if it were a single operation. For example, consider a non-atomic increment ("++A") executed on the same variable by two threads simultaneously:

Thread 1	Thread 2
`reg = A`	`reg = A`
`reg = reg + 1`	`reg = reg + 1`
`A = reg`	`A = reg`

If the threads execute concurrently from top to bottom, both threads will load 0 from A, increment it to 1, and store it back, leaving a final result of 1. If we used an atomic increment operation, you would be guaranteed that the final result will be 2.

Atomic essentials

The most fundamental operations — loading and storing 32-bit values — are inherently atomic on ARM so long

as the data is aligned on a 32-bit boundary. For example:

```
   Thread 1        Thread 2
reg = 0x00000000 reg = 0xffffffff
A = reg          A = reg
```

The CPU guarantees that A will hold 0x00000000 or 0xffffffff. It will never hold 0x0000ffff or any other partial "mix" of bytes.

> *The atomicity guarantee is lost if the data isn't aligned. Misaligned data could straddle a cache line, so other cores could see the halves update independently. Consequently, the ARMv7 documentation declares that it provides "single-copy atomicity" for all byte accesses, halfword accesses to halfword-aligned locations, and word accesses to word-aligned locations. Doubleword (64-bit) accesses are not atomic, unless the location is doubleword-aligned and special load/store instructions are used. This behavior is important to understand when multiple threads are performing unsynchronized updates to packed structures or arrays of primitive types.*

There is no need for 32-bit "atomic read" or "atomic write" functions on ARM or x86. Where one is provided for completeness, it just does a trivial load or store.

Operations that perform more complex actions on data in memory are collectively known as *read-modify-write* (RMW) instructions, because they load data, modify it in some way, and write it back. CPUs vary widely in how these are implemented. ARM uses a technique called "Load Linked / Store Conditional", or LL/SC.

> *A linked or locked load reads the data from memory as usual, but also establishes a reservation,*

> *tagging the physical memory address. The reservation is cleared when another core tries to write to that address. To perform an LL/SC, the data is read with a reservation, modified, and then a conditional store instruction is used to try to write the data back. If the reservation is still in place, the store succeeds; if not, the store will fail. Atomic functions based on LL/SC usually loop, retrying the entire read-modify-write sequence until it completes without interruption.*

It's worth noting that the read-modify-write operations would not work correctly if they operated on stale data. If two cores perform an atomic increment on the same address, and one of them is not able to see what the other did because each core is reading and writing from local cache, the operation won't actually be atomic. The CPU's cache coherency rules ensure that the atomic RMW operations remain atomic in an SMP environment.

This should not be construed to mean that atomic RMW operations use a memory barrier. On ARM, atomics have no memory barrier semantics. While a series of atomic RMW operations on a single address will be observed in program order by other cores, there are no guarantees when it comes to the ordering of atomic and non-atomic operations.

It often makes sense to pair barriers and atomic operations together. The next section describes this in more detail.

Atomic + barrier pairing

As usual, it's useful to illuminate the discussion with an example. We're going to consider a basic mutual-exclusion primitive called a *spin lock*. The idea is that a memory address (which we'll call "lock") initially holds zero. When a thread wants to execute code in the critical section, it sets the lock to 1, executes the critical code, and then changes it back to zero when done. If another thread has already set the lock to 1, we sit and spin until the lock changes back to zero.

To make this work we use an atomic RMW primitive called *compare-and-swap*. The function takes three arguments: the memory address, the expected current value, and the new value. If the value currently in memory matches what we expect, it is replaced with the new value, and the old value is returned. If the current value is not what we expect, we don't change anything. A minor variation on this is called *compare-and-set*; instead of returning the old value it returns a boolean indicating whether the swap succeeded. For our needs either will work, but compare-and-set is slightly simpler for examples, so we use it and just refer to it as "CAS".

The acquisition of the spin lock is written like this (using a C-like language):

```
do {
    success = atomic_cas(&lock, 0, 1)
} while (!success)

full_memory_barrier()

critical-section
```

If no thread holds the lock, the lock value will be 0, and the CAS operation will set it to 1 to indicate that we now have it. If another thread has it, the lock value will be 1, and the CAS operation will fail because the expected current value does not match the actual current value. We loop and retry. (Note this loop is on top of whatever loop the LL/SC code might be doing inside the atomic_cas function.)

> *On SMP, a spin lock is a useful way to guard a small critical section. If we know that another thread is going to execute a handful of instructions and then release the lock, we can just burn a few cycles while we wait our turn. However, if the other thread happens to be executing on the same core, we're just wasting time because the other thread can't make progress until the OS schedules it again (either by migrating it to a different core or by preempting us). A proper spin lock implementation would optimistically spin a few times and then fall back on an OS primitive (such as a Linux futex) that allows the current thread to sleep while waiting for the other thread to finish up. On a uniprocessor you never want to spin at all. For the sake of brevity we're ignoring all*

> *this.*

The memory barrier is necessary to ensure that other threads observe the acquisition of the lock before they observe any loads or stores in the critical section. Without that barrier, the memory accesses could be observed while the lock is not held.

The `full_memory_barrier` call here actually does **two** independent operations. First, it issues the CPU's full barrier instruction. Second, it tells the compiler that it is not allowed to reorder code around the barrier. That way, we know that the `atomic_cas` call will be executed before anything in the critical section. Without this *compiler reorder barrier*, the compiler has a great deal of freedom in how it generates code, and the order of instructions in the compiled code might be much different from the order in the source code.

Of course, we also want to make sure that none of the memory accesses performed in the critical section are observed after the lock is released. The full version of the simple spin lock is:

```
do {
    success = atomic_cas(&lock, 0, 1)   // acquire
} while (!success)
full_memory_barrier()

critical-section

full_memory_barrier()
atomic_store(&lock, 0)                  // release
```

We perform our second CPU/compiler memory barrier immediately **before** we release the lock, so that loads and stores in the critical section are observed before the release of the lock.

As mentioned earlier, the `atomic_store` operation is a simple assignment on ARM and x86. Unlike the atomic RMW operations, we don't guarantee that other threads will see this value immediately. This isn't a problem, though, because we only need to keep the other threads **out**. The other threads will stay out until they observe

the store of 0. If it takes a little while for them to observe it, the other threads will spin a little longer, but we will still execute code correctly.

It's convenient to combine the atomic operation and the barrier call into a single function. It also provides other advantages, which will become clear shortly.

Acquire and release

When acquiring the spinlock, we issue the atomic CAS and then the barrier. When releasing the spinlock, we issue the barrier and then the atomic store. This inspires a particular naming convention: operations followed by a barrier are "acquiring" operations, while operations preceded by a barrier are "releasing" operations. (It would be wise to install the spin lock example firmly in mind, as the names are not otherwise intuitive.)

Rewriting the spin lock example with this in mind:

```
do {
    success = atomic_acquire_cas(&lock, 0, 1)
} while (!success)

critical-section

atomic_release_store(&lock, 0)
```

This is a little more succinct and easier to read, but the real motivation for doing this lies in a couple of optimizations we can now perform.

First, consider `atomic_release_store`. We need to ensure that the store of zero to the lock word is observed after any loads or stores in the critical section above it. In other words, we need a load/store and store/store barrier. In an earlier section we learned that these aren't necessary on x86 SMP -- only store/load barriers are required. The implementation of `atomic_release_store` on x86 is therefore just a compiler reorder barrier followed by a simple store. No CPU barrier is required.

The second optimization mostly applies to the compiler (although some CPUs, such as the Itanium, can take advantage of it as well). The basic principle is that code can move across acquire and release barriers, but only in one direction.

Suppose we have a mix of locally-visible and globally-visible memory accesses, with some miscellaneous computation as well:

```
local1 = arg1 / 41
local2 = threadStruct->field2
threadStruct->field3 = local2

do {
    success = atomic_acquire_cas(&lock, 0, 1)
} while (!success)

local5 = globalStruct->field5
globalStruct->field6 = local5

atomic_release_store(&lock, 0)
```

Here we see two completely independent sets of operations. The first set operates on a thread-local data structure, so we're not concerned about clashes with other threads. The second set operates on a global data structure, which must be protected with a lock.

A full compiler reorder barrier in the atomic ops will ensure that the program order matches the source code order at the lock boundaries. However, allowing the compiler to interleave instructions can improve performance. Loads from memory can be slow, but the CPU can continue to execute instructions that don't require the result of that load while waiting for it to complete. The code might execute more quickly if it were written like this instead:

```
do {
    success = atomic_acquire_cas(&lock, 0, 1)
} while (!success)

local2 = threadStruct->field2
```

```
local5 = globalStruct->field5
local1 = arg1 / 41
threadStruct->field3 = local2
globalStruct->field6 = local5

atomic_release_store(&lock, 0)
```

We issue both loads, do some unrelated computation, and then execute the instructions that make use of the loads. If the integer division takes less time than one of the loads, we essentially get it for free, since it happens during a period where the CPU would have stalled waiting for a load to complete.

Note that **all** of the operations are now happening inside the critical section. Since none of the “threadStruct” operations are visible outside the current thread, nothing else can see them until we’re finished here, so it doesn’t matter exactly when they happen.

In general, it is always safe to move operations **into** a critical section, but never safe to move operations **out of** a critical section. Put another way, you can migrate code “downward” across an acquire barrier, and “upward” across a release barrier. If the atomic ops used a full barrier, this sort of migration would not be possible.

Returning to an earlier point, we can state that on x86 all loads are acquiring loads, and all stores are releasing stores. As a result:

- Loads may not be reordered with respect to each other. You can’t take a load and move it “upward” across another load’s acquire barrier.
- Stores may not be reordered with respect to each other, because you can’t move a store “downward” across another store’s release barrier.
- A load followed by a store can’t be reordered, because neither instruction will tolerate it.
- A store followed by a load **can** be reordered, because each instruction can move across the other in that direction.

Hence, you only need store/load barriers on x86 SMP.

Labeling atomic operations with “acquire” or “release” describes not only whether the barrier is executed before or after the atomic operation, but also how the compiler is allowed to reorder code.

Practice

Debugging memory consistency problems can be very difficult. If a missing memory barrier causes some code to read stale data, you may not be able to figure out why by examining memory dumps with a debugger. By the time you can issue a debugger query, the CPU cores will have all observed the full set of accesses, and the contents of memory and the CPU registers will appear to be in an “impossible” state.

What not to do in C

Here we present some examples of incorrect code, along with simple ways to fix them. Before we do that, we need to discuss the use of a basic language feature.

C/C++ and "volatile"

When writing single-threaded code, declaring a variable “volatile” can be very useful. The compiler will not omit or reorder accesses to volatile locations. Combine that with the sequential consistency provided by the hardware, and you’re guaranteed that the loads and stores will appear to happen in the expected order.

However, accesses to volatile storage may be reordered with non-volatile accesses, so you have to be careful in multi-threaded uniprocessor environments (explicit compiler reorder barriers may be required). There are no atomicity guarantees, and no memory barrier provisions, so “volatile” doesn’t help you at all in multi-threaded SMP environments. The C and C++ language standards are being updated to address this with built-in atomic operations.

If you think you need to declare something “volatile”, that is a strong indicator that you should be using one of the atomic operations instead.

Examples

In most cases you’d be better off with a synchronization primitive (like a pthread mutex) rather than an atomic operation, but we will employ the latter to illustrate how they would be used in a practical situation.

For the sake of brevity we’re ignoring the effects of compiler optimizations here — some of this code is broken even on uniprocessors — so for all of these examples you must assume that the compiler generates straightforward code (for example, compiled with gcc -O0). The fixes presented here do solve both compiler-reordering and memory-access-ordering issues, but we’re only going to discuss the latter.

```
MyThing* gGlobalThing = NULL;

void initGlobalThing()    // runs in thread 1
{
    MyStruct* thing = malloc(sizeof(*thing));
    memset(thing, 0, sizeof(*thing));
    thing->x = 5;
    thing->y = 10;
    /* initialization complete, publish */
    gGlobalThing = thing;
}

void useGlobalThing()    // runs in thread 2
{
    if (gGlobalThing != NULL) {
        int i = gGlobalThing->x;    // could be 5, 0, or uninitialized data
        ...
    }
}
```

The idea here is that we allocate a structure, initialize its fields, and at the very end we “publish” it by storing it in a global variable. At that point, any other thread can see it, but that’s fine since it’s fully initialized, right? At

least, it would be on x86 SMP or a uniprocessor (again, making the erroneous assumption that the compiler outputs code exactly as we have it in the source).

Without a memory barrier, the store to `gGlobalThing` could be observed before the fields are initialized on ARM. Another thread reading from `thing->x` could see 5, 0, or even uninitialized data.

This can be fixed by changing the last assignment to:

```
    atomic_release_store(&gGlobalThing, thing);
```

That ensures that all other threads will observe the writes in the proper order, but what about reads? In this case we should be okay on ARM, because the address dependency rules will ensure that any loads from an offset of `gGlobalThing` are observed after the load of `gGlobalThing`. However, it's unwise to rely on architectural details, since it means your code will be very subtly unportable. The complete fix also requires a barrier after the load:

```
    MyThing* thing = atomic_acquire_load(&gGlobalThing);
    int i = thing->x;
```

Now we know the ordering will be correct. This may seem like an awkward way to write code, and it is, but that's the price you pay for accessing data structures from multiple threads without using locks. Besides, address dependencies won't always save us:

```
MyThing gGlobalThing;

void initGlobalThing()    // runs in thread 1
{
    gGlobalThing.x = 5;
    gGlobalThing.y = 10;
    /* initialization complete */
    gGlobalThing.initialized = true;
}

void useGlobalThing()    // runs in thread 2
{
```

```
    if (gGlobalThing.initialized) {
        int i = gGlobalThing.x;    // could be 5 or 0
    }
}
```

Because there is no relationship between the `initialized` field and the others, the reads and writes can be observed out of order. (Note global data is initialized to zero by the OS, so it shouldn't be possible to read "random" uninitialized data.)

We need to replace the store with:

```
    atomic_release_store(&gGlobalThing.initialized, true);
```

and replace the load with:

```
    int initialized = atomic_acquire_load(&gGlobalThing.initialized);
```

Another example of the same problem occurs when implementing reference-counted data structures. The reference count itself will be consistent so long as atomic increment and decrement operations are used, but you can still run into trouble at the edges, for example:

```
void RefCounted::release()
{
    int oldCount = atomic_dec(&mRefCount);
    if (oldCount == 1) {    // was decremented to zero
        recycleStorage();
    }
}

void useSharedThing(RefCountedThing sharedThing)
{
    int localVar = sharedThing->x;
    sharedThing->release();
    sharedThing = NULL;    // can't use this pointer any more
    doStuff(localVar);    // value of localVar might be wrong
}
```

The `release()` call decrements the reference count using a barrier-free atomic decrement operation. Because this is an atomic RMW operation, we know that it will work correctly. If the reference count goes to zero, we recycle the storage.

The `useSharedThing()` function extracts what it needs from `sharedThing` and then releases its copy. However, because we didn't use a memory barrier, and atomic and non-atomic operations can be reordered, it's possible for other threads to observe the read of `sharedThing->x` **after** they observe the recycle operation. It's therefore possible for `localVar` to hold a value from "recycled" memory, for example a new object created in the same location by another thread after `release()` is called.

This can be fixed by replacing the call to `atomic_dec()` with `atomic_release_dec()`. The barrier ensures that the reads from `sharedThing` are observed before we recycle the object.

> *In most cases the above won't actually fail, because the "recycle" function is likely guarded by functions that themselves employ barriers (libc heap `free()`/`delete()`, or an object pool guarded by a mutex). If the recycle function used a lock-free algorithm implemented without barriers, however, the above code could fail on ARM SMP.*

What not to do in Java

We haven't discussed some relevant Java language features, so we'll take a quick look at those first.

Java's "synchronized" and "volatile" keywords

The "synchronized" keyword provides the Java language's in-built locking mechanism. Every object has an associated "monitor" that can be used to provide mutually exclusive access.

The implementation of the “synchronized” block has the same basic structure as the spin lock example: it begins with an acquiring CAS, and ends with a releasing store. This means that compilers and code optimizers are free to migrate code into a “synchronized” block. One practical consequence: you must **not** conclude that code inside a synchronized block happens after the stuff above it or before the stuff below it in a function. Going further, if a method has two synchronized blocks that lock the same object, and there are no operations in the intervening code that are observable by another thread, the compiler may perform “lock coarsening” and combine them into a single block.

The other relevant keyword is “volatile”. As defined in the specification for Java 1.4 and earlier, a volatile declaration was about as weak as its C counterpart. The spec for Java 1.5 was updated to provide stronger guarantees, almost to the level of monitor synchronization.

The effects of volatile accesses can be illustrated with an example. If thread 1 writes to a volatile field, and thread 2 subsequently reads from that same field, then thread 2 is guaranteed to see that write and all writes previously made by thread 1. More generally, the writes made by **any** thread up to the point where it writes the field will be visible to thead 2 when it does the read. In effect, writing to a volatile is like a monitor release, and reading from a volatile is like a monitor acquire.

Non-volatile accesses may be reorded with respect to volatile accesses in the usual ways, for example the compiler could move a non-volatile load or store “above” a volatile store, but couldn’t move it “below”. Volatile accesses may not be reordered with respect to each other. The VM takes care of issuing the appropriate memory barriers.

It should be mentioned that, while loads and stores of object references and most primitive types are atomic, `long` and `double` fields are not accessed atomically unless they are marked as volatile. Multi-threaded updates to non-volatile 64-bit fields are problematic even on uniprocessors.

Examples

Here's a simple, incorrect implementation of a monotonic counter: (*Java theory and practice: Managing volatility*).

```
class Counter {
    private int mValue;

    public int get() {
        return mValue;
    }
    public void incr() {
        mValue++;
    }
}
```

Assume `get()` and `incr()` are called from multiple threads, and we want to be sure that every thread sees the current count when `get()` is called. The most glaring problem is that `mValue++` is actually three operations:

1. `reg = mValue`
2. `reg = reg + 1`
3. `mValue = reg`

If two threads execute in `incr()` simultaneously, one of the updates could be lost. To make the increment atomic, we need to declare `incr()` "synchronized". With this change, the code will run correctly in multi-threaded uniprocessor environments.

It's still broken on SMP, however. Different threads might see different results from `get()`, because we're reading the value with an ordinary load. We can correct the problem by declaring `get()` to be synchronized. With this change, the code is obviously correct.

Unfortunately, we've introduced the possibility of lock contention, which could hamper performance. Instead of declaring `get()` to be synchronized, we could declare `mValue` with "volatile". (Note `incr()` must still use `synchronize` `mValue`

.) Now we know that the volatile write to will be visible to any subsequent volatile read of `mValue`. `incr()` will be slightly slower, but `get()` will be faster, so even in the absence of contention this is a win if reads outnumber writes. (See also `AtomicInteger`.)

Here's another example, similar in form to the earlier C examples:

```
class MyGoodies {
    public int x, y;
}
class MyClass {
    static MyGoodies sGoodies;

    void initGoodies() {    // runs in thread 1
        MyGoodies goods = new MyGoodies();
        goods.x = 5;
        goods.y = 10;
        sGoodies = goods;
    }

    void useGoodies() {    // runs in thread 2
        if (sGoodies != null) {
            int i = sGoodies.x;    // could be 5 or 0
            ....
        }
    }
}
```

This has the same problem as the C code, namely that the assignment `sGoodies = goods` might be observed before the initialization of the fields in `goods`. If you declare `sGoodies` with the volatile keyword, you can think about the loads as if they were `atomic_acquire_load()` calls, and the stores as if they were `atomic_release_store()` calls.

(Note that only the `sGoodies` reference itself is volatile. The accesses to the fields inside it are not. The statement `z = sGoodies.x` will perform a volatile load of `MyClass.sGoodies` followed by a non-volatile load of `sGoodies.x`. If you make a local reference `MyGoodies localGoods = sGoodies`, `z = localGoods.x` will not perform any volatile loads.)

A more common idiom in Java programming is the infamous “double-checked locking”:

```
class MyClass {
    private Helper helper = null;

    public Helper getHelper() {
        if (helper == null) {
            synchronized (this) {
                if (helper == null) {
                    helper = new Helper();
                }
            }
        }
        return helper;
    }
}
```

The idea is that we want to have a single instance of a `Helper` object associated with an instance of `MyClass`. We must only create it once, so we create and return it through a dedicated `getHelper()` function. To avoid a race in which two threads create the instance, we need to synchronize the object creation. However, we don’t want to pay the overhead for the “synchronized” block on every call, so we only do that part if `helper` is currently null.

This doesn’t work correctly on uniprocessor systems, unless you’re using a traditional Java source compiler and an interpreter-only VM. Once you add fancy code optimizers and JIT compilers it breaks down. See the “‘Double Checked Locking is Broken’ Declaration” link in the appendix for more details, or Item 71 (“Use lazy initialization judiciously”) in Josh Bloch’s *Effective Java, 2nd Edition.*.

Running this on an SMP system introduces an additional way to fail. Consider the same code rewritten slightly, as if it were compiled into a C-like language (I’ve added a couple of integer fields to represent `Helper’s` constructor activity):

```
if (helper == null) {
    // acquire monitor using spinlock
    while (atomic_acquire_cas(&this.lock, 0, 1) != success)
        ;
```

```
    if (helper == null) {
        newHelper = malloc(sizeof(Helper));
        newHelper->x = 5;
        newHelper->y = 10;
        helper = newHelper;
    }
    atomic_release_store(&this.lock, 0);
}
```

Now the problem should be obvious: the store to `helper` is happening before the memory barrier, which means another thread could observe the non-null value of `helper` before the stores to the `x`/`y` fields.

You could try to ensure that the store to `helper` happens after the `atomic_release_store()` on `this.lock` by rearranging the code, but that won't help, because it's okay to migrate code upward — the compiler could move the assignment back above the `atomic_release_store()` to its original position.

There are two ways to fix this:

1. Do the simple thing and delete the outer check. This ensures that we never examine the value of `helper` outside a synchronized block.
2. Declare `helper` volatile. With this one small change, the code in Example J-3 will work correctly on Java 1.5 and later. (You may want to take a minute to convince yourself that this is true.)

This next example illustrates two important issues when using volatile:

```
class MyClass {
    int data1, data2;
    volatile int vol1, vol2;

    void setValues() {    // runs in thread 1
        data1 = 1;
        vol1 = 2;
        data2 = 3;
    }

    void useValues1() {    // runs in thread 2
```

```
        if (vol1 == 2) {
            int l1 = data1;    // okay
            int l2 = data2;    // wrong
        }
    }
    void useValues2() {    // runs in thread 2
        int dummy = vol2;
        int l1 = data1;    // wrong
        int l2 = data2;    // wrong
    }
```

Looking at `useValues1()`, if thread 2 hasn't yet observed the update to `vol1`, then it can't know if `data1` or `data2` has been set yet. Once it sees the update to `vol1`, it knows that the change to `data1` is also visible, because that was made before `vol1` was changed. However, it can't make any assumptions about `data2`, because that store was performed after the volatile store.

The code in `useValues2()` uses a second volatile field, `vol2`, in an attempt to force the VM to generate a memory barrier. This doesn't generally work. To establish a proper "happens-before" relationship, both threads need to be interacting with the same volatile field. You'd have to know that `vol2` was set after `data1/data2` in thread 1. (The fact that this doesn't work is probably obvious from looking at the code; the caution here is against trying to cleverly "cause" a memory barrier instead of creating an ordered series of accesses.)

What to do

General advice

In C/C++, use the `pthread` operations, like mutexes and semaphores. These include the proper memory barriers, providing correct and efficient behavior on all Android platform versions. Be sure to use them correctly, for example be wary of signaling a condition variable without holding the corresponding mutex.

It's best to avoid using atomic functions directly. Locking and unlocking a pthread mutex require a single atomic operation each if there's no contention, so you're not going to save much by replacing mutex calls with atomic

ops. If you need a lock-free design, you must fully understand the concepts in this entire document before you begin (or, better yet, find an existing code library that is known to be correct on SMP ARM).

Be extremely circumspect with "volatile" in C/C++. It often indicates a concurrency problem waiting to happen.

In Java, the best answer is usually to use an appropriate utility class from the `java.util.concurrent` package. The code is well written and well tested on SMP.

Perhaps the safest thing you can do is make your class immutable. Objects from classes like String and Integer hold data that cannot be changed once the class is created, avoiding all synchronization issues. The book *Effective Java, 2nd Ed.* has specific instructions in “Item 15: Minimize Mutability”. Note in particular the importance of declaring fields “final" (Bloch).

If neither of these options is viable, the Java “synchronized” statement should be used to guard any field that can be accessed by more than one thread. If mutexes won’t work for your situation, you should declare shared fields “volatile”, but you must take great care to understand the interactions between threads. The volatile declaration won’t save you from common concurrent programming mistakes, but it will help you avoid the mysterious failures associated with optimizing compilers and SMP mishaps.

The Java Memory Model guarantees that assignments to final fields are visible to all threads once the constructor has finished — this is what ensures proper synchronization of fields in immutable classes. This guarantee does not hold if a partially-constructed object is allowed to become visible to other threads. It is necessary to follow safe construction practices.(Safe Construction Techniques in Java).

Synchronization primitive guarantees

The pthread library and VM make a couple of useful guarantees: all accesses previously performed by a thread that creates a new thread are observable by that new thread as soon as it starts, and all accesses performed by

a thread that is exiting are observable when a `join()` on that thread returns. This means you don't need any additional synchronization when preparing data for a new thread or examining the results of a joined thread.

Whether or not these guarantees apply to interactions with pooled threads depends on the thread pool implementation.

In C/C++, the pthread library guarantees that any accesses made by a thread before it unlocks a mutex will be observable by another thread after it locks that same mutex. It also guarantees that any accesses made before calling `signal()` or `broadcast()` on a condition variable will be observable by the woken thread.

Java language threads and monitors make similar guarantees for the comparable operations.

Upcoming changes to C/C++

The C and C++ language standards are evolving to include a sophisticated collection of atomic operations. A full matrix of calls for common data types is defined, with selectable memory barrier semantics (choose from relaxed, consume, acquire, release, acq_rel, seq_cst).

See the Further Reading section for pointers to the specifications.

Closing Notes

While this document does more than merely scratch the surface, it doesn't manage more than a shallow gouge. This is a very broad and deep topic. Some areas for further exploration:

- Learn the definitions of *happens-before*, *synchronizes-with*, and other essential concepts from the Java Memory Model. (It's hard to understand what "volatile" really means without getting into this.)
- Explore what compilers are and aren't allowed to do when reordering code. (The JSR-133 spec has some

great examples of legal transformations that lead to unexpected results.)

- Find out how to write immutable classes in Java and C++. (There's more to it than just "don't change anything after construction".)
- Internalize the recommendations in the Concurrency section of *Effective Java, 2nd Edition.* (For example, you should avoid calling methods that are meant to be overridden while inside a synchronized block.)
- Understand what sorts of barriers you can use on x86 and ARM. (And other CPUs for that matter, for example Itanium's acquire/release instruction modifiers.)
- Read through the `java.util.concurrent` and `java.util.concurrent.atomic` APIs to see what's available. Consider using concurrency annotations like `@ThreadSafe` and `@GuardedBy` (from net.jcip.annotations).

The Further Reading section in the appendix has links to documents and web sites that will better illuminate these topics.

Appendix

SMP failure example

This document describes a lot of "weird" things that can, in theory, happen. If you're not convinced that these issues are real, a practical example may be useful.

Bill Pugh's Java memory model web site has a few test programs on it. One interesting test is ReadAfterWrite.java, which does the following:

Thread 1	Thread 2
`for (int i = 0; i < ITERATIONS; i++) {`	`for (int i = 0; i < ITERATIONS; i++) {`
`  a = i;`	`  b = i;`
`  BB[i] = b;`	`  AA[i] = a;`

```
}                                        }
```

Where `a` and `b` are declared as volatile `int` fields, and `AA` and `BB` are ordinary integer arrays.

This is trying to determine if the VM ensures that, after a value is written to a volatile, the next read from that volatile sees the new value. The test code executes these loops a million or so times, and then runs through afterward and searches the results for inconsistencies.

At the end of execution,`AA` and `BB` will be full of gradually-increasing integers. The threads will not run side-by-side in a predictable way, but we can assert a relationship between the array contents. For example, consider this execution fragment:

Thread 1	**Thread 2**
`(initially a == 1534)`	`(initially b == 165)`
`a = 1535`	
`BB[1535] = 165`	
`a = 1536`	
`BB[1536] = 165`	
	`b = 166`
	`AA[166] = 1536`
	`b = 167`
	`AA[167] = 1536`
`a = 1537`	
`BB[1537] = 167`	

(This is written as if the threads were taking turns executing so that it’s more obvious when results from one thread should be visible to the other, but in practice that won’t be the case.)

Look at the assignment of `AA[166]` in thread 2. We are capturing the fact that, at the point where thread 2 was on iteration 166, it can see that thread 1 was on iteration 1536. If we look one step in the future, at thread 1's iteration 1537, we expect to see that thread 1 saw that thread 2 was at iteration 166 (or later). `BB[1537]` holds 167, so it appears things are working.

Now suppose we fail to observe a volatile write to `b`:

```
        Thread 1                Thread 2
(initially a == 1534)
a = 1535                  (initially b == 165)
BB[1535] = 165
a = 1536
BB[1536] = 165

                          b = 166
                          AA[166] = 1536
                          b = 167
a = 1537                  AA[167] = 1536
BB[1537] = 165 // stale b
```

Now, `BB[1537]` holds 165, a smaller value than we expected, so we know we have a problem. Put succinctly, for i=166, BB[AA[i]+1] < i. (This also catches failures by thread 2 to observe writes to `a`, for example if we miss an update and assign `AA[166] = 1535`, we will get `BB[AA[166]+1] == 165`.)

If you run the test program under Dalvik (Android 3.0 "Honeycomb" or later) on an SMP ARM device, it will never fail. If you remove the word "volatile" from the declarations of `a` and `b`, it will consistently fail. The program is testing to see if the VM is providing sequentially consistent ordering for accesses to `a` and `b`, so you will only see correct behavior when the variables are volatile. (It will also succeed if you run the code on a

uniprocessor device, or run it while something else is using enough CPU that the kernel doesn't schedule the test threads on separate cores.)

If you run the modified test a few times you will note that it doesn't fail in the same place every time. The test fails consistently because it performs the operations a million times, and it only needs to see out-of-order accesses once. In practice, failures will be infrequent and difficult to locate. This test program could very well succeed on a broken VM if things just happen to work out.

Implementing synchronization stores

(This isn't something most programmers will find themselves implementing, but the discussion is illuminating.)

Consider once again volatile accesses in Java. Earlier we made reference to their similarities with acquiring loads and releasing stores, which works as a starting point but doesn't tell the full story.

We start with a fragment of Dekker's algorithm. Initially both `flag1` and `flag2` are false:

Thread 1	**Thread 2**
`flag1 = true`	`flag2 = true`
`if (flag2 == false)`	`if (flag1 == false)`
`critical-stuff`	*`critical-stuff`*

`flag1` and `flag2` are declared as volatile boolean fields. The rules for acquiring loads and releasing stores would allow the accesses in each thread to be reordered, breaking the algorithm. Fortunately, the JMM has a few things to say here. Informally:

- A write to a volatile field *happens-before* every subsequent read of that same field. (For this example, it means that if one thread updates a flag, and later on the other thread reads that flag, the reader is

guaranteed to see the write.)

- Every execution has a total order over all volatile field accesses. The order is consistent with program order.

Taken together, these rules say that the volatile accesses in our example must be observable in program order by all threads. Thus, we will never see these threads executing the “critical-stuff” simultaneously.

> *Another way to think about this is in terms of data races. A data race occurs if two accesses to the same memory location by different threads are not ordered, at least one of them stores to the memory location, and at least one of them is not a synchronization action (Boehm and McKenney). The memory model declares that a program free of data races must behave as if executed by a sequentially-consistent machine. Because both `flag1` and `flag2` are volatile, and volatile accesses are considered synchronization actions, there are no data races and this code must execute in a sequentially consistent manner.*

As we saw in an earlier section, we need to insert a store/load barrier between the two operations. The code executed in the VM for a volatile access will look something like this:

volatile load	volatile store
	store/store barrier
`reg = A`	`A = reg`
load/load + load/store barrier	*store/load barrier*

The volatile load is just an acquiring load. The volatile store is similar to a releasing store, but we’ve omitted load/store from the pre-store barrier, and added a store/load barrier afterward.

What we’re really trying to guarantee, though, is that (using thread 1 as an example) the write to flag1 is

observed before the read of flag2. We could issue the store/load barrier before the volatile load instead and get the same result, but because loads tend to outnumber stores it's best to associate it with the store.

On some architectures, it's possible to implement volatile stores with an atomic operation and skip the explicit store/load barrier. On x86, for example, atomics provide a full barrier. The ARM LL/SC operations don't include a barrier, so for ARM we must use explicit barriers.

(Much of this is due to Doug Lea and his "JSR-133 Cookbook for Compiler Writers" page.)

Further reading

Web pages and documents that provide greater depth or breadth. The more generally useful articles are nearer the top of the list.

Shared Memory Consistency Models: A Tutorial
: Written in 1995 by Adve & Gharachorloo, this is a good place to start if you want to dive more deeply into memory consistency models.
http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-95-7.pdf

Memory Barriers
: Nice little article summarizing the issues.
http://en.wikipedia.org/wiki/Memory_barrier

Threads Basics
: An introduction to multi-threaded programming in C++ and Java, by Hans Boehm. Excellent discussion of data races and basic synchronization methods.
http://www.hpl.hp.com/personal/Hans_Boehm/c++mm/threadsintro.html

Java Concurrency In Practice
: Published in 2006, this book covers a wide range of topics in great detail. Highly recommended for anyone writing multi-threaded code in Java.

http://www.javaconcurrencyinpractice.com

JSR-133 (Java Memory Model) FAQ

A gentle introduction to the Java memory model, including an explanation of synchronization, volatile variables, and construction of final fields.

http://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html

Overview of package java.util.concurrent

The documentation for the `java.util.concurrent` package. Near the bottom of the page is a section entitled "Memory Consistency Properties" that explains the guarantees made by the various classes.

`java.util.concurrent` Package Summary

Java Theory and Practice: Safe Construction Techniques in Java

This article examines in detail the perils of references escaping during object construction, and provides guidelines for thread-safe constructors.

http://www.ibm.com/developerworks/java/library/j-jtp0618.html

Java Theory and Practice: Managing Volatility

A nice article describing what you can and can't accomplish with volatile fields in Java.

http://www.ibm.com/developerworks/java/library/j-jtp06197.html

The "Double-Checked Locking is Broken" Declaration

Bill Pugh's detailed explanation of the various ways in which double-checked locking is broken. Includes C/C++ and Java.

http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html

[ARM] Barrier Litmus Tests and Cookbook

A discussion of ARM SMP issues, illuminated with short snippets of ARM code. If you found the examples in this document too un-specific, or want to read the formal description of the DMB instruction, read this. Also describes the instructions used for memory barriers on executable code (possibly useful if you're generating code on the fly).

http://infocenter.arm.com/help/topic/com.arm.doc.genc007826/Barrier_Litmus_Tests_and_Cookbook_A08.pdf

Linux Kernel Memory Barriers

Documentation for Linux kernel memory barriers. Includes some useful examples and ASCII art.
http://www.kernel.org/doc/Documentation/memory-barriers.txt

ISO/IEC JTC1 SC22 WG21 (C++ standards) 14882 (C++ programming language), chapter 29 ("Atomic operations library")

Draft standard for C++ atomic operation features.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3090.pdf
(intro: http://www.hpl.hp.com/techreports/2008/HPL-2008-56.pdf)

ISO/IEC JTC1 SC22 WG14 (C standards) 9899 (C programming language) chapter 7.16 ("Atomics <stdatomic.h>")

Draft standard for ISO/IEC 9899-201x C atomic operation features. (See also n1484 for errata.)
http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1425.pdf

Dekker's algorithm

The "first known correct solution to the mutual exclusion problem in concurrent programming". The wikipedia article has the full algorithm, with a discussion about how it would need to be updated to work with modern optimizing compilers and SMP hardware.
http://en.wikipedia.org/wiki/Dekker's_algorithm

Comments on ARM vs. Alpha and address dependencies

An e-mail on the arm-kernel mailing list from Catalin Marinas. Includes a nice summary of address and control dependencies.
http://linux.derkeiler.com/Mailing-Lists/Kernel/2009-05/msg11811.html

What Every Programmer Should Know About Memory

A very long and detailed article about different types of memory, particularly CPU caches, by Ulrich Drepper.
http://www.akkadia.org/drepper/cpumemory.pdf

Reasoning about the ARM weakly consistent memory model

This paper was written by Chong & Ishtiaq of ARM, Ltd. It attempts to describe the ARM SMP memory model in a rigorous but accessible fashion. The definition of "observability" used here comes from this

paper.
http://portal.acm.org/ft_gateway.cfm?id=1353528&type=pdf&coll=&dl=&CFID=96099715&CFTOKEN=57505711

The JSR-133 Cookbook for Compiler Writers

Doug Lea wrote this as a companion to the JSR-133 (Java Memory Model) documentation. It goes much deeper into the details than most people will need to worry about, but it provides good fodder for contemplation.
http://g.oswego.edu/dl/jmm/cookbook.html

The Semantics of Power and ARM Multiprocessor Machine Code

If you prefer your explanations in rigorous mathematical form, this is a fine place to go next.
http://www.cl.cam.ac.uk/~pes20/weakmemory/draft-ppc-arm.pdf

Next Get started Next class

Best Practices for Security & Privacy

These classes and articles provide information about how to keep your app's data secure.

Security Tips

In this document

1. Storing Data
2. Using Permissions
3. Using Networking
4. Performing Input Validation
5. Handling User Data
6. Using WebView
7. Using Cryptography
8. Using Interprocess Communication
9. Dynamically Loading Code
10. Security in a Virtual Machine
11. Security in Native Code

See also

- Android Security Overview
- Permissions

Android has security features built into the operating system that significantly reduce the frequency and impact of application security issues. The system is designed so you can typically build your apps with default system and file permissions and avoid difficult decisions about security.

Some of the core security features that help you build secure apps include:

- The Android Application Sandbox, which isolates your app data and code execution from other apps.
- An application framework with robust implementations of common

security functionality such as cryptography, permissions, and secure IPC.
- Technologies like ASLR, NX, ProPolice, safe_iop, OpenBSD dlmalloc, OpenBSD calloc, and Linux mmap_min_addr to mitigate risks associated with common memory management errors.
- An encrypted filesystem that can be enabled to protect data on lost or stolen devices.
- User-granted permissions to restrict access to system features and user data.
- Application-defined permissions to control application data on a per-app basis.

Nevertheless, it is important that you be familiar with the Android security best practices in this document. Following these practices as general coding habits will reduce the likelihood of inadvertently introducing security issues that adversely affect your users.

Storing Data

The most common security concern for an application on Android is whether the data that you save on the device is accessible to other apps. There are three fundamental ways to save data on the device:

Using internal storage

By default, files that you create on internal storage are accessible only to your app. This protection is implemented by Android and is sufficient for most applications.

You should generally avoid using the `MODE_WORLD_WRITEABLE` or `MODE_WORLD_READABLE` modes for IPC files because they do not provide the ability to limit data access to particular applications, nor do they provide any control on data format. If you want to share your data with other app processes, you might instead consider using a content provider, which offers read and write permissions to other apps and can make dynamic permission grants on a case-by-case basis.

To provide additional protection for sensitive data, you might choose to encrypt local files using a key that is not directly accessible to the application. For example, a key can be placed in a `KeyStore` and protected with a user password that is not stored on the device. While this does not protect data from a root compromise that can monitor the user inputting the password, it can provide protection for a lost device without file system encryption.

Using external storage

Files created on external storage, such as SD Cards, are globally readable and writable. Because external storage can be removed by the user and also modified by any application, you should not store sensitive information using external storage.

As with data from any untrusted source, you should perform input validation when handling data from external storage. We strongly recommend that you not store executables or class files on external storage prior to dynamic loading. If your app does retrieve executable files from external storage, the files should be signed and cryptographically verified prior to dynamic loading.

Using content providers

Content providers offer a structured storage mechanism that can be limited to your own application or exported to allow access by other applications. If you do not intend to provide other applications with access to your `ContentProvider`, mark them as `android:exported=false` in the application manifest. Otherwise, set the `android:exported` attribute `"true"` to allow other apps to access the stored data.

When creating a `ContentProvider` that will be exported for use by other applications, you can specify a single permission for reading and writing, or distinct permissions for reading and writing within the manifest. We recommend that you limit your permissions to those required to accomplish the task at hand. Keep in mind that it's usually easier to add permissions later to expose new functionality than it is to take them away

and break existing users.

If you are using a content provider for sharing data between only your own apps, it is preferable to use the `android:protectionLevel` attribute set to `"signature"` protection. Signature permissions do not require user confirmation, so they provide a better user experience and more controlled access to the content provider data when the apps accessing the data are signed with the same key.

Content providers can also provide more granular access by declaring the `android:grantUriPermissions` attribute and using the `FLAG_GRANT_READ_URI_PERMISSION` and `FLAG_GRANT_WRITE_URI_PERMISSION` flags in the `Intent` object that activates the component. The scope of these permissions can be further limited by the `<grant-uri-permission element>`.

When accessing a content provider, use parameterized query methods such as `query()`, `update()`, and `delete()` to avoid potential SQL injection from untrusted sources. Note that using parameterized methods is not sufficient if the `selection` argument is built by concatenating user data prior to submitting it to the method.

Do not have a false sense of security about the write permission. Consider that the write permission allows SQL statements which make it possible for some data to be confirmed using creative `WHERE` clauses and parsing the results. For example, an attacker might probe for presence of a specific phone number in a call-log by modifying a row only if that phone number already exists. If the content provider data has predictable structure, the write permission may be equivalent to providing both reading and writing.

Using Permissions

Because Android sandboxes applications from each other, applications must explicitly share resources and data. They do this by declaring the permissions they need for additional capabilities not provided by the basic sandbox, including access to device features such as the camera.

Requesting Permissions

We recommend minimizing the number of permissions that your app requests. Not having access to sensitive permissions reduces the risk of inadvertently misusing those permissions, can improve user adoption, and makes your app less vulnerable for attackers. Generally, if a permission is not required for your app to function, do not request it.

If it's possible to design your application in a way that does not require any permissions, that is preferable. For example, rather than requesting access to device information to create a unique identifier, create a GUID for your application (see the section about Handling User Data). Or, rather than using external storage (which requires permission), store data on the internal storage.

In addition to requesting permissions, your application can use the `<permissions>` to protect IPC that is security sensitive and will be exposed to other applications, such as a `ContentProvider`. In general, we recommend using access controls other than user confirmed permissions where possible because permissions can be confusing for users. For example, consider using the signature protection level on permissions for IPC communication between applications provided by a single developer.

Do not leak permission-protected data. This occurs when your app exposes data over IPC that is only available because it has a specific permission, but does not require that permission of any clients of it's IPC interface. More details on the potential impacts, and frequency of this type of problem is provided in this research paper published at USENIX: http://www.cs.be rkeley.edu/~afelt/felt_usenixsec2011.pdf

Creating Permissions

Generally, you should strive to define as few permissions as possible while satisfying your security requirements. Creating a new permission is relatively uncommon for most applications, because the system-defined permissions cover many situations. Where appropriate, perform access checks using existing permissions.

If you must create a new permission, consider whether you can accomplish your task with a "signature" protection level. Signature permissions are transparent to the user and only allow access by applications signed by the same developer as application performing the permission check.

If you create a permission with the "dangerous" protection level, there are a number of complexities that you need to consider:

- The permission must have a string that concisely expresses to a user the security decision they will be required to make.
- The permission string must be localized to many different languages.
- Users may choose not to install an application because a permission is confusing or perceived as risky.
- Applications may request the permission when the creator of the permission has not been installed.

Each of these poses a significant non-technical challenge for you as the developer while also confusing your users, which is why we discourage the use of the "dangerous" permission level.

Using Networking

Network transactions are inherently risky for security, because it involves transmitting data that is potentially private to the user. People are increasingly aware of the privacy concerns of a mobile device, especially when the device performs network transactions, so it's very important that your app implement all best practices toward keeping the user's data secure at all times.

Using IP Networking

Networking on Android is not significantly different from other Linux environments. The key consideration is making sure that appropriate protocols are used for sensitive data, such as `HttpsURLConnection` for secure web traffic. We prefer use of HTTPS over HTTP anywhere that HTTPS is supported on the server, because mobile devices frequently

connect on networks that are not secured, such as public Wi-Fi hotspots.

Authenticated, encrypted socket-level communication can be easily implemented using the `SSLSocket` class. Given the frequency with which Android devices connect to unsecured wireless networks using Wi-Fi, the use of secure networking is strongly encouraged for all applications that communicate over the network.

We have seen some applications use localhost network ports for handling sensitive IPC. We discourage this approach since these interfaces are accessible by other applications on the device. Instead, you should use an Android IPC mechanism where authentication is possible such as with a `Service`. (Even worse than using loopback is to bind to INADDR_ANY since then your application may receive requests from anywhere.)

Also, one common issue that warrants repeating is to make sure that you do not trust data downloaded from HTTP or other insecure protocols. This includes validation of input in `WebView` and any responses to intents issued against HTTP.

Using Telephony Networking

The SMS protocol was primarily designed for user-to-user communication and is not well-suited for apps that want to transfer data. Due to the limitations of SMS, we strongly recommend the use of Google Cloud Messaging (GCM) and IP networking for sending data messages from a web server to your app on a user device.

Beware that SMS is neither encrypted nor strongly authenticated on either the network or the device. In particular, any SMS receiver should expect that a malicious user may have sent the SMS to your application—Do not rely on unauthenticated SMS data to perform sensitive commands. Also, you should be aware that SMS may be subject to spoofing and/or interception on the network. On the Android-powered device itself, SMS messages are transmitted as broadcast intents, so they may be read or captured by other applications that have the `READ_SMS` permission.

Performing Input Validation

Insufficient input validation is one of the most common security problems affecting applications, regardless of what platform they run on. Android does have platform-level countermeasures that reduce the exposure of applications to input validation issues and you should use those features where possible. Also note that selection of type-safe languages tends to reduce the likelihood of input validation issues.

If you are using native code, then any data read from files, received over the network, or received from an IPC has the potential to introduce a security issue. The most common problems are buffer overflows, use after free, and off-by-one errors. Android provides a number of technologies like ASLR and DEP that reduce the exploitability of these errors, but they do not solve the underlying problem. You can prevent these vulneratbilities by careful handling pointers and managing buffers.

Dynamic, string based languages such as JavaScript and SQL are also subject to input validation problems due to escape characters and script injection.

If you are using data within queries that are submitted to an SQL database or a content provider, SQL injection may be an issue. The best defense is to use parameterized queries, as is discussed in the above section about content providers. Limiting permissions to read-only or write-only can also reduce the potential for harm related to SQL injection.

If you cannot use the security features above, we strongly recommend the use of well-structured data formats and verifying that the data conforms to the expected format. While blacklisting of characters or character-replacement can be an effective strategy, these techniques are error-prone in practice and should be avoided when possible.

Handling User Data

In general, the best approach for user data security is to minimize the use of APIs that access sensitive or personal user data. If you have

access to user data and can avoid storing or transmitting the information, do not store or transmit the data. Finally, consider if there is a way that your application logic can be implemented using a hash or non-reversible form of the data. For example, your application might use the hash of an an email address as a primary key, to avoid transmitting or storing the email address. This reduces the chances of inadvertently exposing data, and it also reduces the chance of attackers attempting to exploit your application.

If your application accesses personal information such as passwords or usernames, keep in mind that some jurisdictions may require you to provide a privacy policy explaining your use and storage of that data. So following the security best practice of minimizing access to user data may also simplify compliance.

You should also consider whether your application might be inadvertently exposing personal information to other parties such as third-party components for advertising or third-party services used by your application. If you don't know why a component or service requires a personal information, don’t provide it. In general, reducing the access to personal information by your application will reduce the potential for problems in this area.

If access to sensitive data is required, evaluate whether that information must be transmitted to a server, or whether the operation can be performed on the client. Consider running any code using sensitive data on the client to avoid transmitting user data.

Also, make sure that you do not inadvertently expose user data to other application on the device through overly permissive IPC, world writable files, or network sockets. This is a special case of leaking permission-protected data, discussed in the Requesting Permissions section.

If a GUID is required, create a large, unique number and store it. Do not use phone identifiers such as the phone number or IMEI which may be associated with personal information. This topic is discussed in more detail in the Android Developer Blog.

Be careful when writing to on-device logs. In Android, logs are a shared resource, and are available to an application with the `READ_LOGS` permission. Even though the phone log data is temporary and erased on reboot, inappropriate logging of user information could inadvertently leak user data to other applications.

Using WebView

Because `WebView` consumes web content that can include HTML and JavaScript, improper use can introduce common web security issues such as cross-site-scripting (JavaScript injection). Android includes a number of mechanisms to reduce the scope of these potential issues by limiting the capability of `WebView` to the minimum functionality required by your application.

If your application does not directly use JavaScript within a `WebView`, do *not* call `setJavaScriptEnabled()`. Some sample code uses this method, which you might repurpose in production application, so remove that method call if it's not required. By default, `WebView` does not execute JavaScript so cross-site-scripting is not possible.

Use `addJavaScriptInterface()` with particular care because it allows JavaScript to invoke operations that are normally reserved for Android applications. If you use it, expose `addJavaScriptInterface()` only to web pages from which all input is trustworthy. If untrusted input is allowed, untrusted JavaScript may be able to invoke Android methods within your app. In general, we recommend exposing `addJavaScriptInterface()` only to JavaScript that is contained within your application APK.

If your application accesses sensitive data with a `WebView`, you may want to use the `clearCache()` method to delete any files stored locally. Server-side headers like `no-cache` can also be used to indicate that an application should not cache particular content.

Devices running platforms older than Android 4.4 (API level 19) use a version of `webkit` that has a number of security issues. As a workaround, if your app is running on these devices, it should confirm that `WebView`

objects display only trusted content. You should also use the updatable security `Provider` object to make sure your app isn't exposed to potential vulnerabilities in SSL, as described in Updating Your Security Provider to Protect Against SSL Exploits. If your application must render content from the open web, consider providing your own renderer so you can keep it up to date with the latest security patches.

Handling Credentials

In general, we recommend minimizing the frequency of asking for user credentials—to make phishing attacks more conspicuous, and less likely to be successful. Instead use an authorization token and refresh it.

Where possible, username and password should not be stored on the device. Instead, perform initial authentication using the username and password supplied by the user, and then use a short-lived, service-specific authorization token.

Services that will be accessible to multiple applications should be accessed using `AccountManager`. If possible, use the `AccountManager` class to invoke a cloud-based service and do not store passwords on the device.

After using `AccountManager` to retrieve an `Account`, `CREATOR` before passing in any credentials, so that you do not inadvertently pass credentials to the wrong application.

If credentials are to be used only by applications that you create, then you can verify the application which accesses the `AccountManager` using `checkSignature()`. Alternatively, if only one application will use the credential, you might use a `KeyStore` for storage.

Using Cryptography

In addition to providing data isolation, supporting full-filesystem encryption, and providing secure communications channels, Android provides a wide array of algorithms for protecting data using

cryptography.

In general, try to use the highest level of pre-existing framework implementation that can support your use case. If you need to securely retrieve a file from a known location, a simple HTTPS URI may be adequate and requires no knowledge of cryptography. If you need a secure tunnel, consider using `HttpsURLConnection` or `SSLSocket`, rather than writing your own protocol.

If you do find yourself needing to implement your own protocol, we strongly recommend that you *not* implement your own cryptographic algorithms. Use existing cryptographic algorithms such as those in the implementation of AES or RSA provided in the `Cipher` class.

Use a secure random number generator, `SecureRandom`, to initialize any cryptographic keys, `KeyGenerator`. Use of a key that is not generated with a secure random number generator significantly weakens the strength of the algorithm, and may allow offline attacks.

If you need to store a key for repeated use, use a mechanism like `KeyStore` that provides a mechanism for long term storage and retrieval of cryptographic keys.

Using Interprocess Communication

Some apps attempt to implement IPC using traditional Linux techniques such as network sockets and shared files. We strongly encourage you to instead use Android system functionality for IPC such as `Intent`, `Binder` or `Messenger` with a `Service`, and `BroadcastReceiver`. The Android IPC mechanisms allow you to verify the identity of the application connecting to your IPC and set security policy for each IPC mechanism.

Many of the security elements are shared across IPC mechanisms. If your IPC mechanism is not intended for use by other applications, set the `android:exported` attribute to `"false"` in the component's manifest element, such as for the `<service>` element. This is useful for applications that consist of multiple processes within the same UID, or if you decide late in development that you do not actually want to expose

functionality as IPC but you don't want to rewrite the code.

If your IPC is intended to be accessible to other applications, you can apply a security policy by using the `<permission>` element. If IPC is between your own separate apps that are signed with the same key, it is preferable to use `"signature"` level permission in the `android:protectionLevel`.

Using intents

Intents are the preferred mechanism for asynchronous IPC in Android. Depending on your application requirements, you might use `sendBroadcast()`, `sendOrderedBroadcast()`, or an explicit intent to a specific application component.

Note that ordered broadcasts can be "consumed" by a recipient, so they may not be delivered to all applications. If you are sending an intent that must be delivered to a specific receiver, then you must use an explicit intent that declares the receiver by nameintent.

Senders of an intent can verify that the recipient has a permission specifying a non-Null permission with the method call. Only applications with that permission will receive the intent. If data within a broadcast intent may be sensitive, you should consider applying a permission to make sure that malicious applications cannot register to receive those messages without appropriate permissions. In those circumstances, you may also consider invoking the receiver directly, rather than raising a broadcast.

Note: Intent filters should not be considered a security feature—components can be invoked with explicit intents and may not have data that would conform to the intent filter. You should perform input validation within your intent receiver to confirm that it is properly formatted for the invoked receiver, service, or activity.

Using services

A `Service` is often used to supply functionality for other applications to

use. Each service class must have a corresponding declaration in its manifest file.

By default, services are not exported and cannot be invoked by any other application. However, if you add any intent filters to the service declaration, then it is exported by default. It's best if you explicitly declare the `android:exported` attribute to be sure it behaves as you'd like. Services can also be protected using the `android:permission` attribute. By doing so, other applications will need to declare a corresponding `<uses-permission>` element in their own manifest to be able to start, stop, or bind to the service.

A service can protect individual IPC calls into it with permissions, by calling `checkCallingPermission()` before executing the implementation of that call. We generally recommend using the declarative permissions in the manifest, since those are less prone to oversight.

Using binder and messenger interfaces

Using `Binder` or `Messenger` is the preferred mechanism for RPC-style IPC in Android. They provide a well-defined interface that enables mutual authentication of the endpoints, if required.

We strongly encourage designing interfaces in a manner that does not require interface specific permission checks. `Binder` and `Messenger` objects are not declared within the application manifest, and therefore you cannot apply declarative permissions directly to them. They generally inherit permissions declared in the application manifest for the `Service` or `Activity` within which they are implemented. If you are creating an interface that requires authentication and/or access controls, those controls must be explicitly added as code in the `Binder` or `Messenger` interface.

If providing an interface that does require access controls, use `checkCallingPermission()` to verify whether the caller has a required permission. This is especially important before accessing a service on behalf of the caller, as the identify of your application is passed to other interfaces. If invoking an interface provided by a `Service`, the

`bindService()` invocation may fail if you do not have permission to access the given service. If calling an interface provided locally by your own application, it may be useful to use the `clearCallingIdentity()` to satisfy internal security checks.

For more information about performing IPC with a service, see Bound Services.

Using broadcast receivers

A `BroadcastReceiver` handles asynchronous requests initiated by an `Intent`.

By default, receivers are exported and can be invoked by any other application. If your `BroadcastReceiver` is intended for use by other applications, you may want to apply security permissions to receivers using the `<receiver>` element within the application manifest. This will prevent applications without appropriate permissions from sending an intent to the `BroadcastReceiver`.

Dynamically Loading Code

We strongly discourage loading code from outside of your application APK. Doing so significantly increases the likelihood of application compromise due to code injection or code tampering. It also adds complexity around version management and application testing. Finally, it can make it impossible to verify the behavior of an application, so it may be prohibited in some environments.

If your application does dynamically load code, the most important thing to keep in mind about dynamically loaded code is that it runs with the same security permissions as the application APK. The user made a decision to install your application based on your identity, and they are expecting that you provide any code run within the application, including code that is dynamically loaded.

The major security risk associated with dynamically loading code is that

the code needs to come from a verifiable source. If the modules are included directly within your APK, then they cannot be modified by other applications. This is true whether the code is a native library or a class being loaded using `DexClassLoader`. We have seen many instances of applications attempting to load code from insecure locations, such as downloaded from the network over unencrypted protocols or from world writable locations such as external storage. These locations could allow someone on the network to modify the content in transit, or another application on a users device to modify the content on the device, respectively.

Security in a Virtual Machine

Dalvik is Android's runtime virtual machine (VM). Dalvik was built specifically for Android, but many of the concerns regarding secure code in other virtual machines also apply to Android. In general, you shouldn't concern yourself with security issues relating to the virtual machine. Your application runs in a secure sandbox environment, so other processes on the system cannnot access your code or private data.

If you're interested in diving deeper on the subject of virtual machine security, we recommend that you familiarize yourself with some existing literature on the subject. Two of the more popular resources are:

- http://www.securingjava.com/toc.html
- https://www.owasp.org/index.php/Java_Security_Resources

This document is focused on the areas which are Android specific or different from other VM environments. For developers experienced with VM programming in other environments, there are two broad issues that may be different about writing apps for Android:

- Some virtual machines, such as the JVM or .net runtime, act as a security boundary, isolating code from the underlying operating system capabilities. On Android, the Dalvik VM is not a security boundary—the application sandbox is implemented at the OS level, so Dalvik can interoperate with native code in the same application

without any security constraints.
- Given the limited storage on mobile devices, it's common for developers to want to build modular applications and use dynamic class loading. When doing this, consider both the source where you retrieve your application logic and where you store it locally. Do not use dynamic class loading from sources that are not verified, such as unsecured network sources or external storage, because that code might be modified to include malicious behavior.

Security in Native Code

In general, we encourage developers to use the Android SDK for application development, rather than using native code with the Android NDK. Applications built with native code are more complex, less portable, and more like to include common memory corruption errors such as buffer overflows.

Android is built using the Linux kernel and being familiar with Linux development security best practices is especially useful if you are going to use native code. Linux security practices are beyond the scope of this document, but one of the most popular resources is "Secure Programming for Linux and Unix HOWTO", available at http://www.dwheeler.com/secure-programs.

An important difference between Android and most Linux environments is the Application Sandbox. On Android, all applications run in the Application Sandbox, including those written with native code. At the most basic level, a good way to think about it for developers familiar with Linux is to know that every application is given a unique UID with very limited permissions. This is discussed in more detail in the Android Security Overview and you should be familiar with application permissions even if you are using native code.

Next Get started Next class

Security with HTTPS and SSL

In this document

1. Concepts
2. An HTTP Example
3. Common Problems Verifying Server Certificates
    1. Unknown certificate authority
    2. Self-signed server certificate
    3. Missing intermediate certificate authority
4. Common Problems with Hostname Verification
5. Warnings About Using SSLSocket Directly
6. Blacklisting
7. Pinning
8. Client Certificates
9. Nogotofail: Network Security Testing

See also

- Android Security Overview
- Permissions

The Secure Sockets Layer (SSL)—now technically known as Transport Layer Security (TLS)—is a common building block for encrypted communications between clients and servers. It's possible that an application might use SSL incorrectly such that malicious entities may be able to intercept an app's data over the network. To help you ensure that this does not happen to your app, this article highlights the common pitfalls when using secure network protocols and addresses some larger concerns about using Public-Key Infrastructure (PKI).

Concepts

In a typical SSL usage scenario, a server is configured with a certificate containing a public key as well as a matching private key. As part of the handshake between an SSL client and server, the server proves it has the private key by signing its certificate with public-key cryptography.

However, anyone can generate their own certificate and private key, so a simple handshake doesn't prove anything about the server other than that the server knows the private key that matches the public key of the certificate. One way to solve this problem is to have the client have a set of one or more certificates it trusts. If the certificate is not in the set, the server is not to be trusted.

There are several downsides to this simple approach. Servers should be able to upgrade to stronger keys over time ("key rotation"), which replaces the public key in the certificate with a new one. Unfortunately, now the client app has to be updated due to what is essentially a server configuration change. This is especially problematic if the server is not under the app developer's control, for example if it is a third party web service. This approach also has issues if the app has to talk to arbitrary servers such as a web browser or email app.

In order to address these downsides, servers are typically configured with certificates from well known issuers called Certificate Authorities (CAs). The host platform generally contains a list of well known CAs that it trusts. As of Android 4.2 (Jelly Bean), Android currently contains over 100 CAs that are updated in each release. Similar to a server, a CA has a certificate and a private key. When issuing a certificate for a server, the CA signs the server certificate using its private key. The client can then verify that the server has a certificate issued by a CA known to the platform.

However, while solving some problems, using CAs introduces another. Because the CA issues certificates for many servers, you still need some way to make sure you are talking to the server you want. To address this, the certificate issued by the CA identifies the server either with a specific name such as *gmail.com* or a wildcarded set of hosts such as **.google.com*.

The following example will make these concepts a little more concrete. In the snippet below from a command line, the `openssl` tool's `s_client` command looks at Wikipedia's server certificate information. It specifies port 443 because that is the default for HTTPS. The command sends the output of `openssl s_client` to `openssl x509`, which formats information about certificates according to the X.509 standard. Specifically, the command asks for the subject, which contains the server name information, and the issuer, which identifies the CA.

```
$ openssl s_client -connect wikipedia.org:443 | openssl x509 -noout
-subject -issuer
subject=
/serialNumber=sOrr2rKpMVP70Z6E9BT5reY008SJEdYv/C=US/O=*.wikipedia.org/C
www.rapidssl.com/resources/cps (c)11/OU=Domain Control Validated -
RapidSSL(R)/CN=*.wikipedia.org
issuer= /C=US/O=GeoTrust, Inc./CN=RapidSSL CA
```

You can see that the certificate was issued for servers matching **.wikipedia.org* by the RapidSSL CA.

An HTTPS Example

Assuming you have a web server with a certificate issued by a well known CA, you can make a secure request with code as simple this:

```
URL url = new URL("https://wikipedia.org");
URLConnection urlConnection = url.openConnection();
InputStream in = urlConnection.getInputStream();
copyInputStreamToOutputStream(in, System.out);
```

Yes, it really can be that simple. If you want to tailor the HTTP request, you can cast to an `HttpURLConnection`. The Android documentation for `HttpURLConnection` has further examples about how to deal with request and response headers, posting content, managing cookies, using proxies, caching responses, and so on. But in terms of the details for verifying certificates and hostnames, the Android framework takes care of it for you through these APIs. This is where you want to be if at all possible. That said, below are some other considerations.

Common Problems Verifying Server Certificates

Suppose instead of receiving the content from `getInputStream()`, it throws an exception:

```
javax.net.ssl.SSLHandshakeException:
java.security.cert.CertPathValidatorException: Trust anchor for
certification path not found.
        at
org.apache.harmony.xnet.provider.jsse.OpenSSLSocketImpl.startHandshake(

        at
libcore.net.http.HttpConnection.setupSecureSocket(HttpConnection.java:2

        at
libcore.net.http.HttpsURLConnectionImpl$HttpsEngine.makeSslConnection(H

        at
libcore.net.http.HttpsURLConnectionImpl$HttpsEngine.connect(HttpsURLCon

        at
libcore.net.http.HttpEngine.sendSocketRequest(HttpEngine.java:290)
        at
libcore.net.http.HttpEngine.sendRequest(HttpEngine.java:240)
        at
libcore.net.http.HttpURLConnectionImpl.getResponse(HttpURLConnectionImp

        at
libcore.net.http.HttpURLConnectionImpl.getInputStream(HttpURLConnection

        at
libcore.net.http.HttpsURLConnectionImpl.getInputStream(HttpsURLConnecti
```

This can happen for several reasons, including:

1. The CA that issued the server certificate was unknown
2. The server certificate wasn't signed by a CA, but was self signed
3. The server configuration is missing an intermediate CA

The following sections discuss how to address these problems while keeping your connection to the server secure.

Unknown certificate authority

In this case, the `SSLHandshakeException` occurs because you have a CA that isn't trusted by the system. It could be because you have a certificate from a new CA that isn't yet trusted by Android or your app is running on an older version without the CA. More often a CA is unknown because it isn't a public CA, but a private one issued by an organization

such as a government, corporation, or education institution for their own use.

Fortunately, you can teach `HttpsURLConnection` to trust a specific set of CAs. The procedure can be a little convoluted, so below is an example that takes a specific CA from an `InputStream`, uses it to create a `KeyStore`, which is then used to create and initialize a `TrustManager`. A `TrustManager` is what the system uses to validate certificates from the server and—by creating one from a `KeyStore` with one or more CAs—those will be the only CAs trusted by that `TrustManager`.

Given the new `TrustManager`, the example initializes a new `SSLContext` which provides an `SSLSocketFactory` you can use to override the default `SSLSocketFactory` from `HttpsURLConnection`. This way the connection will use your CAs for certificate validation.

Here is the example in full using an organizational CA from the University of Washington:

```
// Load CAs from an InputStream
// (could be from a resource or ByteArrayInputStream or ...)
CertificateFactory cf = CertificateFactory.getInstance("X.509");
// From https://www.washington.edu/itconnect/security/ca/load-der.crt
InputStream caInput = new BufferedInputStream(new
FileInputStream("load-der.crt"));
Certificate ca;
try {
    ca = cf.generateCertificate(caInput);
    System.out.println("ca=" + ((X509Certificate)
ca).getSubjectDN());
} finally {
    caInput.close();
}

// Create a KeyStore containing our trusted CAs
String keyStoreType = KeyStore.getDefaultType();
KeyStore keyStore = KeyStore.getInstance(keyStoreType);
keyStore.load(null, null);
keyStore.setCertificateEntry("ca", ca);

// Create a TrustManager that trusts the CAs in our KeyStore
String tmfAlgorithm = TrustManagerFactory.getDefaultAlgorithm();
TrustManagerFactory tmf =
TrustManagerFactory.getInstance(tmfAlgorithm);
tmf.init(keyStore);

// Create an SSLContext that uses our TrustManager
SSLContext context = SSLContext.getInstance("TLS");
context.init(null, tmf.getTrustManagers(), null);
```

```
// Tell the URLConnection to use a SocketFactory from our SSLContext
URL url = new URL("https://certs.cac.washington.edu/CAtest/");
HttpsURLConnection urlConnection =
    (HttpsURLConnection)url.openConnection();
urlConnection.setSSLSocketFactory(context.getSocketFactory());
InputStream in = urlConnection.getInputStream();
copyInputStreamToOutputStream(in, System.out);
```

With a custom `TrustManager` that knows about your CAs, the system is able to validate that your server certificate come from a trusted issuer.

Caution: Many web sites describe a poor alternative solution which is to install a `TrustManager` that does nothing. If you do this you might as well not be encrypting your communication, because anyone can attack your users at a public Wi-Fi hotspot by using DNS tricks to send your users' traffic through a proxy of their own that pretends to be your server. The attacker can then record passwords and other personal data. This works because the attacker can generate a certificate and—without a `TrustManager` that actually validates that the certificate comes from a trusted source—your app could be talking to anyone. So don't do this, not even temporarily. You can always make your app trust the issuer of the server's certificate, so just do it.

Self-signed server certificate

The second case of `SSLHandshakeException` is due to a self-signed certificate, which means the server is behaving as its own CA. This is similar to an unknown certificate authority, so you can use the same approach from the previous section.

You can create your own `TrustManager`, this time trusting the server certificate directly. This has all of the downsides discussed earlier of tying your app directly to a certificate, but can be done securely. However, you should be careful to make sure your self-signed certificate has a reasonably strong key. As of 2012, a 2048-bit RSA signature with an exponent of 65537 expiring yearly is acceptable. When rotating keys, you should check for recommendations from an authority (such as NIST) about what is acceptable.

Missing intermediate certificate authority

The third case of `SSLHandshakeException` occurs due to a missing intermediate CA. Most public CAs don't sign server certificates directly. Instead, they use their main CA certificate, referred to as the root CA, to sign intermediate CAs. They do this so the root CA can be stored offline to reduce risk of compromise. However, operating systems like Android typically trust only root CAs directly, which leaves a short gap of trust between the server certificate—signed by the intermediate CA—and the certificate verifier, which knows the root CA. To solve this, the server doesn't send the client only it's certificate during the SSL handshake, but a chain of certificates from the server CA through any intermediates necessary to reach a trusted root CA.

To see what this looks like in practice, here's the *mail.google.com* certificate chain as viewed by the `openssl` `s_client` command:

```
$ openssl s_client -connect mail.google.com:443
---
Certificate chain
 0 s:/C=US/ST=California/L=Mountain View/O=Google
Inc/CN=mail.google.com
   i:/C=ZA/O=Thawte Consulting (Pty) Ltd./CN=Thawte SGC CA
 1 s:/C=ZA/O=Thawte Consulting (Pty) Ltd./CN=Thawte SGC CA
   i:/C=US/O=VeriSign, Inc./OU=Class 3 Public Primary Certification
Authority
---
```

This shows that the server sends a certificate for *mail.google.com* issued by the *Thawte SGC* CA, which is an intermediate CA, and a second certificate for the *Thawte SGC* CA issued by a *Verisign* CA, which is the primary CA that's trusted by Android.

However, it is not uncommon to configure a server to not include the necessary intermediate CA. For example, here is a server that can cause an error in Android browsers and exceptions in Android apps:

```
$ openssl s_client -connect egov.uscis.gov:443
---
Certificate chain
 0 s:/C=US/ST=District Of Columbia/L=Washington/O=U.S. Department of
Homeland Security/OU=United States Citizenship and Immigration
Services/OU=Terms of use at www.verisign.com/rpa
(c)05/CN=egov.uscis.gov
   i:/C=US/O=VeriSign, Inc./OU=VeriSign Trust Network/OU=Terms of
use at https://www.verisign.com/rpa (c)10/CN=VeriSign Class 3
International Server CA - G3
---
```

What is interesting to note here is that visiting this server in most desktop browsers does not cause an error like a completely unknown CA or self-signed server certificate would cause. This is because most desktop browsers cache trusted intermediate CAs over time. Once a browser has visited and learned about an intermediate CA from one site, it won't need to have the intermediate CA included in the certificate chain the next time.

Some sites do this intentionally for secondary web servers used to serve resources. For example, they might have their main HTML page served by a server with a full certificate chain, but have servers for resources such as images, CSS, or JavaScript not include the CA, presumably to save bandwidth. Unfortunately, sometimes these servers might be providing a web service you are trying to call from your Android app, which is not as forgiving.

There are two approaches to solve this issue:

- Configure the server to include the intermediate CA in the server chain. Most CAs provide documentation on how to do this for all common web servers. This is the only approach if you need the site to work with default Android browsers at least through Android 4.2.
- Or, treat the intermediate CA like any other unknown CA, and create a `TrustManager` to trust it directly, as done in the previous two sections.

Common Problems with Hostname Verification

As mentioned at the beginning of this article, there are two key parts to verifying an SSL connection. The first is to verify the certificate is from a trusted source, which was the focus of the previous section. The focus of this section is the second part: making sure the server you are talking to presents the right certificate. When it doesn't, you'll typically see an error like this:

```
java.io.IOException: Hostname 'example.com' was not verified
```

```
        at
libcore.net.http.HttpConnection.verifySecureSocketHostname(HttpConnecti

        at
libcore.net.http.HttpsURLConnectionImpl$HttpsEngine.connect(HttpsURLCon

        at
libcore.net.http.HttpEngine.sendSocketRequest(HttpEngine.java:290)
        at
libcore.net.http.HttpEngine.sendRequest(HttpEngine.java:240)
        at
libcore.net.http.HttpURLConnectionImpl.getResponse(HttpURLConnectionImp

        at
libcore.net.http.HttpURLConnectionImpl.getInputStream(HttpURLConnectior

        at
libcore.net.http.HttpsURLConnectionImpl.getInputStream(HttpsURLConnecti
```

One reason this can happen is due to a server configuration error. The server is configured with a certificate that does not have a subject or subject alternative name fields that match the server you are trying to reach. It is possible to have one certificate be used with many different servers. For example, looking at the *google.com* certificate with `openssl s_client -connect google.com:443 | openssl x509 -text` you can see that a subject that supports **.google.com* but also subject alternative names for **.youtube.com*, **.android.com*, and others. The error occurs only when the server name you are connecting to isn't listed by the certificate as acceptable.

Unfortunately this can happen for another reason as well: virtual hosting. When sharing a server for more than one hostname with HTTP, the web server can tell from the HTTP/1.1 request which target hostname the client is looking for. Unfortunately this is complicated with HTTPS, because the server has to know which certificate to return before it sees the HTTP request. To address this problem, newer versions of SSL, specifically TLSv.1.0 and later, support Server Name Indication (SNI), which allows the SSL client to specify the intended hostname to the server so the proper certificate can be returned.

Fortunately, `HttpsURLConnection` supports SNI since Android 2.3. Unfortunately, Apache HTTP Client does not, which is one of the many reasons we discourage its use. One workaround if you need to support Android 2.2 (and older) or Apache HTTP Client is to set up an alternative

virtual host on a unique port so that it's unambiguous which server certificate to return.

The more drastic alternative is to replace `HostnameVerifier` with one that uses not the hostname of your virtual host, but the one returned by the server by default.

Caution: Replacing `HostnameVerifier` can be **very dangerous** if the other virtual host is not under your control, because a man-in-the-middle attack could direct traffic to another server without your knowledge.

If you are still sure you want to override hostname verification, here is an example that replaces the verifier for a single `URLConnection` with one that still verifies that the hostname is at least on expected by the app:

```
// Create an HostnameVerifier that hardwires the expected hostname.
// Note that is different than the URL's hostname:
// example.com versus example.org
HostnameVerifier hostnameVerifier = new HostnameVerifier() {
    @Override
    public boolean verify(String hostname, SSLSession session) {
        HostnameVerifier hv =
            HttpsURLConnection.getDefaultHostnameVerifier();
        return hv.verify("example.com", session);
    }
};

// Tell the URLConnection to use our HostnameVerifier
URL url = new URL("https://example.org/");
HttpsURLConnection urlConnection =
    (HttpsURLConnection)url.openConnection();
urlConnection.setHostnameVerifier(hostnameVerifier);
InputStream in = urlConnection.getInputStream();
copyInputStreamToOutputStream(in, System.out);
```

But remember, if you find yourself replacing hostname verification, especially due to virtual hosting, it's still **very dangerous** if the other virtual host is not under your control and you should find an alternative hosting arrangement that avoids this issue.

Warnings About Using SSLSocket Directly

So far, the examples have focused on HTTPS using `HttpsURLConnection`. Sometimes apps need to use SSL separate from HTTP. For example, an email app might use SSL variants of SMTP, POP3, or IMAP. In those

cases, the app would want to use `SSLSocket` directly, much the same way that `HttpsURLConnection` does internally.

The techniques described so far to deal with certificate verification issues also apply to `SSLSocket`. In fact, when using a custom `TrustManager`, what is passed to `HttpsURLConnection` is an `SSLSocketFactory`. So if you need to use a custom `TrustManager` with an `SSLSocket`, follow the same steps and use that `SSLSocketFactory` to create your `SSLSocket`.

Caution: `SSLSocket` **does not** perform hostname verification. It is up the your app to do its own hostname verification, preferably by calling `getDefaultHostnameVerifier()` with the expected hostname. Further beware that `HostnameVerifier.verify()` doesn't throw an exception on error but instead returns a boolean result that you must explicitly check.

Here is an example showing how you can do this. It shows that when connecting to *gmail.com* port 443 without SNI support, you'll receive a certificate for *mail.google.com*. This is expected in this case, so check to make sure that the certificate is indeed for *mail.google.com*:

```
// Open SSLSocket directly to gmail.com
SocketFactory sf = SSLSocketFactory.getDefault();
SSLSocket socket = (SSLSocket) sf.createSocket("gmail.com", 443);
HostnameVerifier hv =
HttpsURLConnection.getDefaultHostnameVerifier();
SSLSession s = socket.getSession();

// Verify that the certicate hostname is for mail.google.com
// This is due to lack of SNI support in the current SSLSocket.
if (!hv.verify("mail.google.com", s)) {
    throw new SSLHandshakeException("Expected mail.google.com, "
                                    "found " + s.getPeerPrincipal());
}

// At this point SSLSocket performed certificate verificaiton and
// we have performed hostname verification, so it is safe to
proceed.

// ... use socket ...
socket.close();
```

Blacklisting

SSL relies heavily on CAs to issue certificates to only the properly verified owners of servers and domains. In rare cases, CAs are either tricked or,

in the case of Comodo or DigiNotar, breached, resulting in the certificates for a hostname to be issued to someone other than the owner of the server or domain.

In order to mitigate this risk, Android has the ability to blacklist certain certificates or even whole CAs. While this list was historically built into the operating system, starting in Android 4.2 this list can be remotely updated to deal with future compromises.

Pinning

An app can further protect itself from fraudulently issued certificates by a technique known as pinning. This is basically using the example provided in the unknown CA case above to restrict an app's trusted CAs to a small set known to be used by the app's servers. This prevents the compromise of one of the other 100+ CAs in the system from resulting in a breach of the apps secure channel.

Client Certificates

This article has focused on the user of SSL to secure communications with servers. SSL also supports the notion of client certificates that allow the server to validate the identity of a client. While beyond the scope of this article, the techniques involved are similar to specifying a custom `TrustManager`. See the discussion about creating a custom `KeyManager` in the documentation for `HttpsURLConnection`.

Nogotofail: A Network Traffic Security Testing Tool

Nogotofail is a tool gives you an easy way to confirm that your apps are safe against known TLS/SSL vulnerabilities and misconfigurations. It's an automated, powerful, and scalable tool for testing network security issues on any device whose network traffic could be made to go through it.

Nogotofail is useful for three main use cases:

- Finding bugs and vulnerabilities.
- Verifying fixes and watching for regressions.
- Understanding what applications and devices are generating what traffic.

Nogotofail works for Android, iOS, Linux, Windows, Chrome OS, OSX, in fact any device you use to connect to the Internet. There's an easy-to-use client to configure the settings and get notifications on Android and Linux, as well as the attack engine itself which can be deployed as a router, VPN server, or proxy.

You can access the tool at the Nogotofail open source project.

Next Get started Next class

Updating Your Security Provider to Protect Against SSL Exploits

In this document

1. Patching the Security Provider with ProviderInstaller
2. Patching Synchronously
3. Patching Asynchronously

See also

- Google Play Services
- OpenSSL Security Advisory [05 Jun 2014]: SSL/TLS MITM vulnerability (CVE-2014-0224)
- Vulnerability Summary for CVE-2014-0224

Android relies on a security `Provider` to provide secure network communications. However, from time to time, vulnerabilities are found in the default security provider. To protect against these vulnerabilities, Google Play services provides a way to automatically update a device's security provider to protect against known exploits. By calling Google Play services methods, your app can ensure that it's running on a device that has the latest updates to protect against known exploits.

For example, a vulnerability was discovered in OpenSSL (CVE-2014-0224) that can leave apps open to a "man-in-the-middle" attack that decrypts secure traffic without either side knowing. With Google Play services version 5.0, a fix is available, but apps must ensure that this fix is installed. By using the Google Play services methods, your app can ensure that it's running on a device that's secured against that attack.

Caution: Updating a device's security `Provider` does *not* update `android.net.SSLCertificateSocketFactory`. Rather than using this class,

we encourage app developers to use high-level methods for interacting with cryptography. Most apps can use APIs like `HttpsURLConnection` without needing to set a custom `TrustManager` or create an `SSLCertificateSocketFactory`.

Patching the Security Provider with ProviderInstaller

To update a device's security provider, use the `ProviderInstaller` class. You can verify that the security provider is up-to-date (and update it, if necessary) by calling that class's `installIfNeeded()` (or `installIfNeededAsync()`) method.

When you call `installIfNeeded()`, the `ProviderInstaller` does the following:

- If the device's `Provider` is successfully updated (or is already up-to-date), the method returns normally.
- If the device's Google Play services library is out of date, the method throws `GooglePlayServicesRepairableException`. The app can then catch this exception and show the user an appropriate dialog box to update Google Play services.
- If a non-recoverable error occurs, the method throws `GooglePlayServicesNotAvailableException` to indicate that it is unable to update the `Provider`. The app can then catch the exception and choose an appropriate course of action, such as displaying the standard fix-it flow diagram.

The `installIfNeededAsync()` method behaves similarly, except that instead of throwing exceptions, it calls the appropriate callback method to indicate success or failure.

If `installIfNeeded()` needs to install a new `Provider`, this can take anywhere from 30-50 milliseconds (on more recent devices) to 350 ms (on older devices). If the security provider is already up-to-date, the method takes a negligible amount of time. To avoid affecting user experience:

- Call `installIfNeeded()` from background networking threads immediately when the threads are loaded, instead of waiting for the thread to try to use the network. (There's no harm in calling the method multiple times, since it returns immediately if the security provider doesn't need updating.)
- If user experience will be affected by the thread blocking--for example, if the call is from an activity in the UI thread--call the asynchronous version of the method, `installIfNeededAsync()`. (Of course, if you do this, you need to wait for the operation to finish before you attempt any secure communications. The `ProviderInstaller` calls your listener's `onProviderInstalled()` method to signal success.)

Warning: If the `ProviderInstaller` is unable to install an updated `Provider`, your device's security provider might be vulnerable to known exploits. Your app should behave as if all HTTP communication is unencrypted.

Once the `Provider` is updated, all calls to security APIs (including SSL APIs) are routed through it. (However, this does not apply to `android.net.SSLCertificateSocketFactory`, which remains vulnerable to such exploits as CVE-2014-0224.)

Patching Synchronously

The simplest way to patch the security provider is to call the synchronous method `installIfNeeded()`. This is appropriate if user experience won't be affected by the thread blocking while it waits for the operation to finish.

For example, here's an implementation of a sync adapter that updates the security provider. Since a sync adapter runs in the background, it's okay if the thread blocks while waiting for the security provider to be updated. The sync adapter calls `installIfNeeded()` to update the security provider. If the method returns normally, the sync adapter knows the security provider is up-to-date. If the method throws an exception, the sync adapter can take appropriate action (such as prompting the user to

update Google Play services).

```
/**
 * Sample sync adapter using {@link ProviderInstaller}.
 */
public class SyncAdapter extends AbstractThreadedSyncAdapter {

  ...

  // This is called each time a sync is attempted; this is okay,
since the
  // overhead is negligible if the security provider is up-to-date.
  @Override
  public void onPerformSync(Account account, Bundle extras, String
authority,
      ContentProviderClient provider, SyncResult syncResult) {
    try {
      ProviderInstaller.installIfNeeded(getContext());
    } catch (GooglePlayServicesRepairableException e) {

      // Indicates that Google Play services is out of date,
disabled, etc.

      // Prompt the user to install/update/enable Google Play
services.
      GooglePlayServicesUtil.showErrorNotification(
          e.getConnectionStatusCode(), getContext());

      // Notify the SyncManager that a soft error occurred.
      syncResult.stats.numIOExceptions++;
      return;

    } catch (GooglePlayServicesNotAvailableException e) {
      // Indicates a non-recoverable error; the ProviderInstaller is
not able
      // to install an up-to-date Provider.

      // Notify the SyncManager that a hard error occurred.
      syncResult.stats.numAuthExceptions++;
      return;
    }

    // If this is reached, you know that the provider was already
up-to-date,
    // or was successfully updated.
  }
}
```

Patching Asynchronously

Updating the security provider can take as much as 350 milliseconds (on older devices). If you're doing the update on a thread that directly affects user experience, such as the UI thread, you don't want to make a synchronous call to update the provider, since that can result in the app

or device freezing until the operation finishes. Instead, you should use the asynchronous method `installIfNeededAsync()`. That method indicates its success or failure by calling callbacks.

For example, here's some code that updates the security provider in an activity in the UI thread. The activity calls `installIfNeededAsync()` to update the provider, and designates itself as the listener to receive success or failure notifications. If the security provider is up-to-date or is successfully updated, the activity's `onProviderInstalled()` method is called, and the activity knows communication is secure. If the provider cannot be updated, the activity's `onProviderInstallFailed()` method is called, and the activity can take appropriate action (such as prompting the user to update Google Play services).

```
/**
 * Sample activity using {@link ProviderInstaller}.
 */
public class MainActivity extends Activity
    implements ProviderInstaller.ProviderInstallListener {

  private static final int ERROR_DIALOG_REQUEST_CODE = 1;

  private boolean mRetryProviderInstall;

  //Update the security provider when the activity is created.
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    ProviderInstaller.installIfNeededAsync(this, this);
  }

  /**
   * This method is only called if the provider is successfully
updated
   * (or is already up-to-date).
   */
  @Override
  protected void onProviderInstalled() {
    // Provider is up-to-date, app can make secure network calls.
  }

  /**
   * This method is called if updating fails; the error code
indicates
   * whether the error is recoverable.
   */
  @Override
  protected void onProviderInstallFailed(int errorCode, Intent
recoveryIntent) {
    if (GooglePlayServicesUtil.isUserRecoverableError(errorCode)) {
      // Recoverable error. Show a dialog prompting the user to
      // install/update/enable Google Play services.
```

```
      GooglePlayServicesUtil.showErrorDialogFragment(
          errorCode,
          this,
          ERROR_DIALOG_REQUEST_CODE,
          new DialogInterface.OnCancelListener() {
            @Override
            public void onCancel(DialogInterface dialog) {
              // The user chose not to take the recovery action
              onProviderInstallerNotAvailable();
            }
          });
    } else {
      // Google Play services is not available.
      onProviderInstallerNotAvailable();
    }
  }

  @Override
  protected void onActivityResult(int requestCode, int resultCode,
      Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    if (requestCode == ERROR_DIALOG_REQUEST_CODE) {
      // Adding a fragment via
GooglePlayServicesUtil.showErrorDialogFragment
      // before the instance state is restored throws an error. So
instead,
      // set a flag here, which will cause the fragment to delay
until
      // onPostResume.
      mRetryProviderInstall = true;
    }
  }

  /**
   * On resume, check to see if we flagged that we need to reinstall
the
   * provider.
   */
  @Override
  protected void onPostResume() {
    super.onPostResult();
    if (mRetryProviderInstall) {
      // We can now safely retry installation.
      ProviderInstall.installIfNeededAsync(this, this);
    }
    mRetryProviderInstall = false;
  }

  private void onProviderInstallerNotAvailable() {
    // This is reached if the provider cannot be updated for some
reason.
    // App should consider all HTTP communication to be vulnerable,
and take
    // appropriate action.
  }
}
```

Next Get started Next class

Checking Device Compatibility with SafetyNet

Previous Next Get started

In this document

1. Additional Terms of Service
2. Connect to Play Services
3. Requesting a Compatibility Check
    1. Obtain Single Use Token
    2. Send Compatibility Check Request
    3. Read Compatibility Check Response
    4. Verify Compatibility Check Response

SafetyNet provides services for analyzing the configuration of a particular device, to make sure that apps function properly on a particular device and that users have a great experience.

The service provides an API your app can use to analyze the device where it is installed. The API uses software and hardware information on the device where your app is installed to create a profile of that device. The service then attempts to match it to a list of device models that have passed Android compatibility testing. This check can help you decide if the device is configured in a way that is consistent with the Android platform specifications and has the capabilities to run your app.

This document shows you how to use SafetyNet for analyzing a device and help you determine if your app will function as expected on that device.

Additional Terms of Service

By accessing or using the SafetyNet APIs, you agree to the Google APIs Terms of Service, and to these Additional Terms. Please read and understand all applicable terms and policies before accessing the APIs.

SafetyNet Terms of Service

As with any data collected in large volume from in-the-field observation, there is a chance of both false positives and false negatives. We are presenting the data to the best of our understanding. We extensively test our detection mechanisms to ensure accuracy, and we are committed to improving those methods over time to ensure they continue to remain accurate. You agree to comply with all applicable law, regulation, and third party rights (including without limitation laws regarding the import or export of data or software, privacy, and local laws). You will not use the APIs to encourage or promote illegal activity or violation of third party rights. You will not violate any other terms of service with Google (or its affiliates). You acknowledge and understand that the SafetyNet API works by collecting hardware and software information, such as device and application data and the results of integrity checks, and sending that data to Google for analysis. Pursuant to Section 3(d) of the Google APIs Terms of Service, you agree that if you use the APIs that it is your responsibility to provide any necessary notices or consents for the collection and sharing of this data with Google.

Connect to Google Play Services

The SafetyNet API is part of Google Play services. To connect to the API, you need to create an instance of the Google Play services API client. For details about using the client in your app, see Accessing Google APIs. Once you have established a connection to Google Play services, you can use the Google API client classes to connect to the SafetyNet API.

To connect to the API, in your activity's onCreate() method, create an instance of Google API Client using `GoogleApiClient.Builder`. Use the builder to add the SafetyNet API, as shown in the following code example:

```
protected synchronized void buildGoogleApiClient() {
    mGoogleApiClient = new GoogleApiClient.Builder(this)
            .addApi(SafetyNet.API)
            .addConnectionCallbacks(myMainActivity.this)
            .build();
}
```

Note: You can only call these methods after your app has established a connection to Google Play services by receiving the `onConnected()` callback. For details about listening for a completed client connection, see Accessing Google APIs.

Requesting a Compatibility Check

A SafetyNet compatibility check allows your app to check if the device where it is running matches the profile of a device that has passed Android compatibility testing. The compatibility check creates a device profile by gathering information about the device hardware and software characteristics, including the platform build.

Using the API to perform a check requires a few implementation steps in your app. Once you have established a connection to Google Play services and requested the SafetyNet API from the Google API client, your app can then perform the following steps to use the service:

- Obtain a single use token
- Send the compatibility check request
- Read the response
- Validate the response

For more information about Android compatibility testing, see Android Compatibility and the Compatibility Testing Suite (CTS).

SafetyNet checks use network resources, and so the speed of responses to requests can vary, depending on a device's network connection status. The code described in this section should be executed outside of your app's main execution thread, to avoid pauses and unresponsiveness in your app user interface. For more information about using separate execution threads, see Sending Operations to Multiple Threads.

Obtain a single use token

The SafetyNet API uses security techniques to help you verify the integrity of the communications between your app and the service. When you request a compatibility check, you must provide a single use token in the form of a number used once, or *nonce*, as part of your request. A nonce is a random token generated in a cryptographically secure manner.

You can obtain a nonce by generating one within your app each time you make a compatibility check request. As a more secure option, you can obtain a nonce from your own server, using a secure connection.

A nonce used with a SafetyNet request should be at least 16 bytes in length. After you make a check request, the response from the SafetyNet service includes your nonce, so you can verify it against the one you sent. As the name indicates, you should only use a nonce value once, for a single check request. Use a different nonce for any subsequent check requests. For tips on using cryptography functions, see Security Tips.

Send the compatibility check request

After you have established a connection to Google Play services and created a nonce, you are ready to make a compatibility check request. Since the response to your request may not be immediate, you set up a callback listener to catch the response from the service, as shown in the following code example:

```
byte[] nonce = getRequestNonce(); // Should be at least 16 bytes in
length.
SafetyNet.SafetyNetApi.attest(mGoogleApiClient, nonce)
        .setResultCallback(new
ResultCallback<SafetyNetApi.AttestationResult>() {

    @Override
    public void onResult(SafetyNetApi.AttestationResult result) {
        Status status = result.getStatus();
        if (status.isSuccess()) {
            // Indicates communication with the service was successful.
            // result.getJwsResult() contains the result data
        } else {
            // An error occurred while communicating with the service
        }
    }
});
```

The `isSuccess()` method indicates whether or not communication with the service was successful, but does not indicate if the device has passed the compatibility check. The next section discusses how to read the check result and verify its integrity.

Read the compatibility check response

When your app communicates with SafetyNet, the service provides a response containing the result and additional information to help you verify the integrity of the message. The result is provided as a `AttestationResult` object. Use the `getJwsResult()` method of this object to obtain the data of the request. The response is formatted as a JSON Web Signature (JWS), the following JWS excerpt shows the format of the payload data:

```
{
"nonce": "R2Rra24fVm5xa2Mg",
"timestampMs": 9860437986543,
"apkPackageName": "com.package.name.of.requesting.app",
"apkCertificateDigestSha256": ["base64 encoded, SHA-256 hash of the
certificate used to sign requesting app"],
"apkDigestSha256": "base64 encoded, SHA-256 hash of the app's APK",
"ctsProfileMatch": true,
}
```

If the value of `ctsProfileMatch` is `true`, this indicates that the device profile matches a device that has passed Android compatibility testing. If the output of the `getJwsResult()` method is null or contains an `error:` field, then communication with the service failed and should be retried. You should use an exponential backoff technique for retries, to avoid flooding the service with additional requests.

Verify the compatibility check response

You should take steps to make sure the response received by your app actually came from the SafetyNet service and matches the request data you provided. Follow these steps to verify the origin of the JWS message:

- Extract the SSL certificate chain from the JWS message.
- Validate the SSL certificate chain and use SSL hostname matching to verify that the leaf certificate was issued to the hostname `attest.android.com`.
- Use the certificate to verify the signature of the JWS message.

After completing this validation, you should also check the data of the JWS message to make sure it matches your original request, including the nonce, timestamp, package name, and the SHA-256 hashes. You can perform these validation steps within your app, or as a more secure option, send the entire JWS response to your own server for verification, via a secure connection.

Validating the response with Google APIs

Google provides an Android Device Verification API for validating the output of the SafetyNet compatibility check. This API performs a validation check on the JWS message returned from the SafetyNet service.

To enable access to the Android Device Verification API:

1. Go to the Google Developers Console.

2. Select a project, or create a new one.
3. In the sidebar on the left, expand **APIs & auth**. Next, click **APIs**. In the list of APIs, make sure all of the APIs you are using show a status of **ON**.
4. In the **Browse APIs** list, find the **Android Device Verification API** and turn it on.
5. Obtain your API key by expanding **APIs & auth** and clicking **Credentials**. Record the **API KEY** (*not* the *Android Key*) value on this page for later use.

After enabling this API for your project, you can call the verification service from your app or server. You need the contents of the JWS message from the SafetyNet API and your API key to call the verification API and get a result.

To use the Android Device Verification API:

1. Create a JSON message containing the entire contents of the JWS message in the following format:

   ```
   { "signedAttestation": "<output of getJwsResult()>" }
   ```

2. Use an HTTP POST request to send the message with a Content-Type of `"application/json"` to the following URL:

   ```
   https://www.googleapis.com/androidcheck/v1/attestations/verify?
   key=<your API key>
   ```

3. The service validates the integrity of the message, and if the message is valid, it returns a JSON message with the following contents:

   ```
   { “isValidSignature”: true }
   ```

Important: This use of the Android Device Verification API only validates that the provided JWS message was received from the SafetyNet service. It *does not* verify that the payload data matches your original compatibility check request.

Next Get started Next class

Enhancing Security with Device Management Policies

Previous Next Get started

This lesson teaches you to

1. Define and Declare Your Policy
2. Create a Device Administration Receiver
3. Activate the Device Administrator
4. Implement the Device Policy Controller

You should also read

- Device Administration

Try it out

Download the sample

DeviceManagement.zip

Since Android 2.2 (API level 8), the Android platform offers system-level device management capabilities through the Device Administration APIs.

In this lesson, you will learn how to create a security-aware application that manages access to its content by enforcing device management policies. Specifically, the application can be configured such that it ensures a screen-lock password of sufficient strength is set up before displaying restricted content to the user.

Define and Declare Your Policy

First, you need to define the kinds of policy to support at the functional level. Policies may cover screen-lock password strength, expiration timeout, encryption, etc.

You must declare the selected policy set, which will be enforced by the application, in the `res/xml/device_admin.xml` file. The Android manifest should also reference the declared policy set.

Each declared policy corresponds to some number of related device policy methods in `DevicePolicyManager` (defining minimum password length and minimum number of uppercase characters are two examples). If an application attempts to invoke methods whose corresponding policy is not declared in the XML, this will result in a `SecurityException` at runtime. Other permissions, such as `force-lock`, are available if the application intends to manage other kinds of policy. As you'll see later, as part of the device administrator activation process, the list of declared policies will be presented to the user on a system screen.

The following snippet declares the limit password policy in `res/xml/device_admin.xml`:

```
<device-admin
xmlns:android="http://schemas.android.com/apk/res/android">
    <uses-policies>
        <limit-password />
    </uses-policies>
</device-admin>
```

Policy declaration XML referenced in Android manifest:

```
<receiver android:name=".Policy$PolicyAdmin"
    android:permission="android.permission.BIND_DEVICE_ADMIN">
    <meta-data android:name="android.app.device_admin"
        android:resource="@xml/device_admin" />
    <intent-filter>
        <action
android:name="android.app.action.DEVICE_ADMIN_ENABLED" />
    </intent-filter>
</receiver>
```

Create a Device Administration Receiver

Create a Device Administration broadcast receiver, which gets notified of

events related to the policies you've declared to support. An application can selectively override callback methods.

In the sample application, Device Admin, when the device administrator is deactivated by the user, the configured policy is erased from the shared preference. You should consider implementing business logic that is relevant to your use case. For example, the application might take some actions to mitigate security risk by implementing some combination of deleting sensitive data on the device, disabling remote synchronization, alerting an administrator, etc.

For the broadcast receiver to work, be sure to register it in the Android manifest as illustrated in the above snippet.

```
public static class PolicyAdmin extends DeviceAdminReceiver {

    @Override
    public void onDisabled(Context context, Intent intent) {
        // Called when the app is about to be deactivated as a
device administrator.
        // Deletes previously stored password policy.
        super.onDisabled(context, intent);
        SharedPreferences prefs =
context.getSharedPreferences(APP_PREF, Activity.MODE_PRIVATE);
        prefs.edit().clear().commit();
    }
}
```

Activate the Device Administrator

Before enforcing any policies, the user needs to manually activate the application as a device administrator. The snippet below illustrates how to trigger the settings activity in which the user can activate your application. It is good practice to include the explanatory text to highlight to users why the application is requesting to be a device administrator, by specifying the `EXTRA_ADD_EXPLANATION` extra in the intent.

Figure 1. The user activation screen in which you can provide a description of your device

policies.

```
if (!mPolicy.isAdminActive()) {

    Intent activateDeviceAdminIntent =
        new Intent(DevicePolicyManager.ACTION_ADD_DEVICE_ADMIN);

    activateDeviceAdminIntent.putExtra(
        DevicePolicyManager.EXTRA_DEVICE_ADMIN,
        mPolicy.getPolicyAdmin());

    // It is good practice to include the optional explanation text
to
    // explain to user why the application is requesting to be a
device
    // administrator. The system will display this message on the
activation
    // screen.
    activateDeviceAdminIntent.putExtra(
        DevicePolicyManager.EXTRA_ADD_EXPLANATION,

getResources().getString(R.string.device_admin_activation_message));

    startActivityForResult(activateDeviceAdminIntent,
        REQ_ACTIVATE_DEVICE_ADMIN);
}
```

If the user chooses "Activate," the application becomes a device administrator and can begin configuring and enforcing the policy.

The application also needs to be prepared to handle set back situations where the user abandons the activation process by hitting the Cancel button, the Back key, or the Home key. Therefore, `onResume()` in the Policy Set Up Activity needs to have logic to reevaluate the condition and present the Device Administrator Activation option to the user if needed.

Implement the Device Policy Controller

After the device administrator is activated successfully, the application then configures Device Policy Manager with the requested policy. Keep in mind that new policies are being added to Android with each release. It is appropriate to perform version checks in your application if using new policies while supporting older versions of the platform. For example, the Password Minimum Upper Case policy is only available with API level 11 (Honeycomb) and above. The following code demonstrates how you can check the version at runtime.

```
DevicePolicyManager mDPM = (DevicePolicyManager)
        context.getSystemService(Context.DEVICE_POLICY_SERVICE);
ComponentName mPolicyAdmin = new ComponentName(context,
PolicyAdmin.class);
...
mDPM.setPasswordQuality(mPolicyAdmin,
PASSWORD_QUALITY_VALUES[mPasswordQuality]);
mDPM.setPasswordMinimumLength(mPolicyAdmin, mPasswordLength);
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
    mDPM.setPasswordMinimumUpperCase(mPolicyAdmin,
mPasswordMinUpperCase);
}
```

At this point, the application is able to enforce the policy. While the application has no access to the actual screen-lock password used, through the Device Policy Manager API it can determine whether the existing password satisfies the required policy. If it turns out that the existing screen-lock password is not sufficient, the device administration API does not automatically take corrective action. It is the application's responsibility to explicitly launch the system password-change screen in the Settings app. For example:

```
if (!mDPM.isActivePasswordSufficient()) {
    ...
    // Triggers password change screen in Settings.
    Intent intent =
        new Intent(DevicePolicyManager.ACTION_SET_NEW_PASSWORD);
    startActivity(intent);
}
```

Normally, the user can select from one of the available lock mechanisms, such as None, Pattern, PIN (numeric), or Password (alphanumeric). When a password policy is configured, those password types that are weaker than those defined in the policy are disabled. For example, if the "Numeric" password quality is configured, the user can select either PIN (numeric) or Password (alphanumeric) password only.

Once the device is properly secured by setting up a proper screen-lock password, the application allows access to the secured content.

```
if (!mDPM.isAdminActive(..)) {
    // Activates device administrator.
    ...
} else if (!mDPM.isActivePasswordSufficient()) {
    // Launches password set-up screen in Settings.
    ...
} else {
    // Grants access to secure content.
    ...
```

```
    startActivity(new Intent(context, SecureActivity.class));
}
```

Next Get started Next class

Best Practices for Testing

These classes and articles provide information about how to test your Android application.

Testing Your Android Activity

Previous Next Get started

Dependencies and prerequisites

- Android 2.2 (API Level 8) or higher.

You Should Also Read

- Testing (Developer's Guide)

You should be writing and running tests as part of your Android application development cycle. Well-written tests can help you to catch bugs early in development and give you confidence in your code.

A *test case* defines a set of objects and methods to run multiple tests independently from each other. Test cases can be organized into *test suites* and run programmatically, in a repeatable manner, with a *test runner* provided by a testing framework.

The lessons in this class teaches you how to use the Android's custom testing framework that is based on the popular JUnit framework. You can write test cases to verify specific behavior in your application, and check for consistency across different Android devices. Your test cases also serve as a form of internal code documentation by describing the expected behavior of app components.

Lessons

Setting Up Your Test Environment
Learn how to create your test project.
Creating and Running a Test Case

Learn how to write test cases to verify the expected properties of your `Activity`, and run the test cases with the `Instrumentation` test runner provided by the Android framework.

Testing UI Components

Learn how to test the behavior of specific UI components in your `Activity`.

Creating Unit Tests

Learn how to how to perform unit testing to verify the behavior of an Activity in isolation.

Creating Functional Tests

Learn how to perform functional testing to verify the interaction of multiple Activities.

Next Get started Next class

Setting Up Your Test Environment

Previous Next Get started

This lesson teaches you to

1. Set Up Eclipse for Testing
2. Set Up the Command Line Interface for Testing

You should also read

- Getting the SDK Bundle
- Testing from Eclipse with ADT
- Testing from Other IDEs

Try it out

Download the demo

AndroidTestingFun.zip

Before you start writing and running your tests, you should set up your test development environment. This lesson teaches you how to set up the Eclipse IDE to build and run tests, and how to build and run tests with the Gradle framework by using the command line interface.

Note: To help you get started, the lessons are based on Eclipse with the ADT plugin. However, for your own test development, you are free to use the IDE of your choice or the command-line.

Set Up Eclipse for Testing

Eclipse with the Android Developer Tools (ADT) plugin provides an integrated development environment for you to create, build, and run Android application test cases from a graphical user interface (GUI). A convenient feature that Eclipse provides is the ability to auto-generate a new test project that corresponds with your Android application project.

To set up your test environment in Eclipse:

1. Download and install the Eclipse ADT plugin, if you haven't installed it yet.
2. Import or create the Android application project that you want to test against.
3. Generate a test project that corresponds to the application project under test. To generate a test project for the app project that you imported:

    a. In the Package Explorer, right-click on your app project, then select **Android Tools** > **New Test Project**.
    b. In the New Android Test Project wizard, set the property values for your test project then click **Finish**.

You should now be able to create, build, and run test cases from your Eclipse environment. To learn how to perform these tasks in Eclipse, proceed to Creating and Running a Test Case.

Set Up the Command Line Interface for Testing

If you are using Gradle version 1.6 or higher as your build environment, you can build and run your Android application tests from the command line by using the Gradle Wrapper. Make sure that in your `gradle.build` file, the minSdkVersion attribute in the `defaultConfig` section is set to 8 or higher. You can refer to the sample `gradle.build` file that is included in the download bundle for this training class.

To run your tests with the Gradle Wrapper:

1. Connect a physical Android device to your machine or launch the Android Emulator.
2. Run the following command from your project directory:

   ```
   ./gradlew build connectedCheck
   ```

To learn more about using Gradle for Android testing, see the Gradle Plugin User Guide.

To learn more about using command line tools other than Gradle for test development, see Testing from Other IDEs.

Next Get started Next class

Creating and Running a Test Case

Previous Next Get started

This lesson teaches you to

1. Create a Test Case for Activity Testing
    1. Set Up Your Test Fixture
    2. Add Test Preconditions
    3. Add Test Methods to Verify Your Activity
2. Build and Run Your Test

You should also read

- Testing Fundamentals

In order to verify that there are no regressions in the layout design and functional behavior in your application, it's important to create a test for each `Activity` in your application. For each test, you need to create the individual parts of a test case, including the test fixture, preconditions test method, and `Activity` test methods. You can then run your test to get a test report. If any test method fails, this might indicate a potential defect in your code.

Note: In the Test-Driven Development (TDD) approach, instead of writing most or all of your app code up-front and then running tests later in the development cycle, you would progressively write just enough production code to satisfy your test dependencies, update your test cases to reflect new functional requirements, and iterate repeatedly this way.

Create a Test Case

`Activity` tests are written in a structured way. Make sure to put your tests in a separate package, distinct from the code under test.

By convention, your test package name should follow the same name as the application package, suffixed with **".tests"**. In the test package you created, add the Java class for your test case. By convention, your test case name should also follow the same name as the Java or Android class that you want to test, but suffixed with **"Test"**.

To create a new test case in Eclipse:

a. In the Package Explorer, right-click on the `/src` directory for your test project and select **New > Package**.
b. Set the **Name** field to `<your_app_package_name>.tests` (for example, `com.example.android.testingfun.tests`) and click **Finish**.
c. Right-click on the test package you created, and select **New > Class**.
d. Set the **Name** field to `<your_app_activity_name>Test` (for example, `MyFirstTestActivityTest`) and click **Finish**.

Set Up Your Test Fixture

A *test fixture* consists of objects that must be initialized for running one or more tests. To set up the test fixture, you can override the `setUp()` and `tearDown()` methods in your test. The test runner automatically runs `setUp()` before running any other test methods, and `tearDown()` at the end of each test method execution. You can use these methods to keep the code for test initialization and clean up separate from the tests methods.

To set up your test fixture in Eclipse:

1. In the Package Explorer, double-click on the test case that you created earlier to bring up the Eclipse Java editor, then modify your test case class to extend one of the sub-classes of `ActivityTestCase`.

   For example:

```
public class MyFirstTestActivityTest
        extends
ActivityInstrumentationTestCase2<MyFirstTestActivity> {
```

2. Next, add the constructor and `setUp()` methods to your test case, and add variable declarations for the `Activity` that you want to test.

   For example:

```
public class MyFirstTestActivityTest
        extends
ActivityInstrumentationTestCase2<MyFirstTestActivity> {

    private MyFirstTestActivity mFirstTestActivity;
    private TextView mFirstTestText;

    public MyFirstTestActivityTest() {
        super(MyFirstTestActivity.class);
    }

    @Override
    protected void setUp() throws Exception {
        super.setUp();
        mFirstTestActivity = getActivity();
        mFirstTestText =
                (TextView) mFirstTestActivity
                .findViewById(R.id.my_first_test_text_view);
    }
}
```

   The constructor is invoked by the test runner to instantiate the test class, while the `setUp()` method is invoked by the test runner before it runs any tests in the test class.

Typically, in the `setUp()` method, you should:

- Invoke the superclass constructor for `setUp()`, which is required by JUnit.
- Initialize your test fixture state by:
  - Defining the instance variables that store the state of the fixture.
  - Creating and storing a reference to an instance of the `Activity` under test.
  - Obtaining a reference to any UI components in the `Activity` that you want to test.

You can use the `getActivity()` method to get a reference to the `Activity`

under test.

Add Test Preconditions

As a sanity check, it is good practice to verify that the test fixture has been set up correctly, and the objects that you want to test have been correctly instantiated or initialized. That way, you won't have to see tests failing because something was wrong with the setup of your test fixture. By convention, the method for verifying your test fixture is called `testPreconditions()`.

For example, you might want to add a `testPreconditons()` method like this to your test case:

```
public void testPreconditions() {
    assertNotNull("mFirstTestActivity is null", mFirstTestActivity);
    assertNotNull("mFirstTestText is null", mFirstTestText);
}
```

The assertion methods are from the JUnit `Assert` class. Generally, you can use assertions to verify if a specific condition that you want to test is true.

- If the condition is false, the assertion method throws an `AssertionFailedError` exception, which is then typically reported by the test runner. You can provide a string in the first argument of your assertion method to give some contextual details if the assertion fails.
- If the condition is true, the test passes.

In both cases, the test runner proceeds to run the other test methods in the test case.

Add Test Methods to Verify Your Activity

Next, add one or more test methods to verify the layout and functional behavior of your `Activity`.

For example, if your `Activity` includes a `TextView`, you can add a test method like this to check that it has the correct label text:

```
public void testMyFirstTestTextView_labelText() {
    final String expected =
            mFirstTestActivity.getString(R.string.my_first_test);
    final String actual = mFirstTestText.getText().toString();
    assertEquals(expected, actual);
}
```

The `testMyFirstTestTextView_labelText()` method simply checks that the default text of the `TextView` that is set by the layout is the same as the expected text defined in the `strings.xml` resource.

Note: When naming test methods, you can use an underscore to separate what is being tested from the specific case being tested. This style makes it easier to see exactly what cases are being tested.

When doing this type of string value comparison, it's good practice to read the expected string from your resources, instead of hardcoding the string in your comparison code. This prevents your test from easily breaking whenever the string definitions are modified in the resource file.

To perform the comparison, pass both the expected and actual strings as arguments to the `assertEquals()` method. If the values are not the same, the assertion will throw an `AssertionFailedError` exception.

If you added a `testPreconditions()` method, put your test methods after the `testPreconditions()` definition in your Java class.

For a complete test case example, take a look at `MyFirstTestActivityTest.java` in the sample app.

Build and Run Your Test

You can build and run your test easily from the Package Explorer in Eclipse.

To build and run your test:

1. Connect an Android device to your machine. On the device or emulator, open the **Settings** menu, select **Developer options** and make sure that USB debugging is enabled.

2. In the Project Explorer, right-click on the test class that you created earlier and select **Run As > Android Junit Test**.
3. In the Android Device Chooser dialog, select the device that you just connected, then click **OK**.
4. In the JUnit view, verify that the test passes with no errors or failures.

For example, if the test case passes with no errors, the result should look like this:

Figure 1. Result of a test with no errors.

Next Get started Next class

Testing UI Components

Previous Next Get started

This lesson teaches you to

1. Create a Test Case for UI Testing with Instrumentation
2. Add Test Methods to Verify UI Behavior
    1. Verify Button Layout Parameters
    2. Verify TextView Layout Parameters
    3. Verify Button Behavior
3. Apply Test Annotations

Try it out

Download the demo

AndroidTestingFun.zip

Typically, your `Activity` includes user interface components (such as buttons, editable text fields, checkboxes, and pickers) to allow users to interact with your Android application. This lesson shows how you can test an `Activity` with a simple push-button UI. You can use the same general steps to test other, more sophisticated types of UI components.

Note: The type of UI testing in this lesson is called *white-box testing* because you have the source code for the application that you want to test. The Android Instrumentation framework is suitable for creating white-box tests for UI components within an application. An alternative type of UI testing is *black-box testing*, where you may not have access to the application source. This type of testing is useful when you want to test how your app interacts with other apps or with the system. Black-box testing is not covered in this training. To learn more about how to

perform black-box testing on your Android apps, see the UI Testing guide.

For a complete test case example, take a look at `ClickFunActivityTest.java` in the sample app.

Create a Test Case for UI Testing with Instrumentation

When testing an `Activity` that has a user interface (UI), the `Activity` under test runs in the UI thread. However, the test application itself runs in a separate thread in the same process as the application under test. This means that your test app can reference objects from the UI thread, but if it attempts to change properties on those objects or send events to the UI thread, you will usually get a `WrongThreadException` error.

To safely inject `Intent` objects into your `Activity` or run test methods on the UI thread, you can extend your test class to use `ActivityInstrumentationTestCase2`. To learn more about how to run test methods on the UI thread, see Testing on the UI thread.

Set Up Your Test Fixture

When setting up the test fixture for UI testing, you should specify the touch mode in your `setUp()` method. Setting the touch mode to `true` prevents the UI control from taking focus when you click it programmatically in the test method later (for example, a button UI will just fire its on-click listener). Make sure that you call `setActivityInitialTouchMode()` before calling `getActivity()`.

For example:

```
public class ClickFunActivityTest
        extends ActivityInstrumentationTestCase2 {
    ...
    @Override
    protected void setUp() throws Exception {
        super.setUp();

        setActivityInitialTouchMode(true);
```

```
        mClickFunActivity = getActivity();
        mClickMeButton = (Button)
                mClickFunActivity
                .findViewById(R.id.launch_next_activity_button);
        mInfoTextView = (TextView)
                mClickFunActivity.findViewById(R.id.info_text_view);
    }
}
```

Add Test Methods to Validate UI Behavior

Your UI testing goals might include:

- Verifying that a button is displayed with the correct layout when the `Activity` is launched.
- Verifying that a `TextView` is initially hidden.
- Verifying that a `TextView` displays the expected string when a button is pushed.

The following section demonstrates how you can implement test methods to perform these verifications.

Verify Button Layout Parameters

You might add a test method like this to verify that a button is displayed correctly in your `Activity`:

```
@MediumTest
public void testClickMeButton_layout() {
    final View decorView =
mClickFunActivity.getWindow().getDecorView();

    ViewAsserts.assertOnScreen(decorView, mClickMeButton);

    final ViewGroup.LayoutParams layoutParams =
            mClickMeButton.getLayoutParams();
    assertNotNull(layoutParams);
    assertEquals(layoutParams.width,
WindowManager.LayoutParams.MATCH_PARENT);
    assertEquals(layoutParams.height,
WindowManager.LayoutParams.WRAP_CONTENT);
}
```

In the `assertOnScreen()` method call, you should pass in the root view and the view that you are expecting to be present on the screen. If the expected view is not found in the root view, the assertion method throws

an `AssertionFailedError` exception, otherwise the test passes.

You can also verify that the layout of a `Button` is correct by getting a reference to its `ViewGroup.LayoutParams` object, then call assertion methods to verify that the `Button` object's width and height attributes match the expected values.

The `@MediumTest` annotation specifies how the test is categorized, relative to its absolute execution time. To learn more about using test size annotations, see Apply Test Annotations.

Verify TextView Layout Parameters

You might add a test method like this to verify that a `TextView` initially appears hidden in your `Activity`:

```
@MediumTest
public void testInfoTextView_layout() {
    final View decorView =
mClickFunActivity.getWindow().getDecorView();
    ViewAsserts.assertOnScreen(decorView, mInfoTextView);
    assertTrue(View.GONE == mInfoTextView.getVisibility());
}
```

You can call `getDecorView()` to get a reference to the decor view for the `Activity`. The decor view is the top-level ViewGroup (`FrameLayout`) view in the layout hierarchy.

Verify Button Behavior

You can use a test method like this to verify that a `TextView` becomes visible when a `Button` is pushed:

```
@MediumTest
public void testClickMeButton_clickButtonAndExpectInfoText() {
    String expectedInfoText =
mClickFunActivity.getString(R.string.info_text);
    TouchUtils.clickView(this, mClickMeButton);
    assertTrue(View.VISIBLE == mInfoTextView.getVisibility());
    assertEquals(expectedInfoText, mInfoTextView.getText());
}
```

To programmatically click a `Button` in your test, call `clickView()`. You must pass in a reference to the test case that is being run and a

reference to the `Button` to manipulate.

Note: The `TouchUtils` helper class provides convenience methods for simulating touch interactions with your application. You can use these methods to simulate clicking, tapping, and dragging of Views or the application screen.

Caution: The `TouchUtils` methods are designed to send events to the UI thread safely from the test thread. You should not run `TouchUtils` directly in the UI thread or any test method annotated with `@UIThread`. Doing so might raise the `WrongThreadException`.

Apply Test Annotations

The following annotations can be applied to indicate the size of a test method:

`@SmallTest`
: Marks a test that should run as part of the small tests.

`@MediumTest`
: Marks a test that should run as part of the medium tests.

`@LargeTest`
: Marks a test that should run as part of the large tests.

Typically, a short running test that take only a few milliseconds should be marked as a `@SmallTest`. Longer running tests (100 milliseconds or more) are usually marked as `@MediumTest`s or `@LargeTest`s, depending on whether the test accesses resources on the local system only or remote resources over a network. For guidance on using test size annotations, see this Android Tools Protip.

You can mark up your test methods with other test annotations to control how the tests are organized and run. For more information on other annotations, see the `Annotation` class reference.

Next Get started Next class

Creating Unit Tests

Previous Next Get started

This lesson teaches you to

1. Create a Test Case for Activity Unit Testing
2. Validate Launch of Another Activity

Try it out

Download the demo

AndroidTestingFun.zip

An `Activity` unit test is an excellent way to quickly verify the state of an `Activity` and its interactions with other components in isolation (that is, disconnected from the rest of the system). A unit test generally tests the smallest possible unit of code (which could be a method, class, or component), without dependencies on system or network resources. For example, you can write a unit test to check that an `Activity` has the correct layout or that it triggers an `Intent` object correctly.

Unit tests are generally not suitable for testing complex UI interaction events with the system. Instead, you should use the `ActivityInstrumentationTestCase2` class, as described in Testing UI Components.

This lesson shows how you can write a unit test to verify that an `Intent` is triggered to launch another `Activity`. Since the test runs in an isolated environment, the `Intent` is not actually sent to the Android system, but you can inspect that the `Intent` object's payload data is accurate.

For a complete test case example, take a look at

`LaunchActivityTest.java` in the sample app.

Note: To test against system or external dependencies, you can use mock objects from a mocking framework and inject them into your unit tests. To learn more about the mocking framework provided by Android, see Mock Object Classes.

Create a Test Case for Activity Unit Testing

The `ActivityUnitTestCase` class provides support for isolated testing of a single `Activity`. To create a unit test for your `Activity`, your test class should extend `ActivityUnitTestCase`.

The `Activity` in an `ActivityUnitTestCase` is not automatically started by Android Instrumentation. To start the `Activity` in isolation, you need to explicitly call the `startActivity()` method, and pass in the `Intent` to launch your target `Activity`.

For example:

```
public class LaunchActivityTest
        extends ActivityUnitTestCase<LaunchActivity> {
    ...

    @Override
    protected void setUp() throws Exception {
        super.setUp();
        mLaunchIntent = new Intent(getInstrumentation()
                .getTargetContext(), LaunchActivity.class);
        startActivity(mLaunchIntent, null, null);
        final Button launchNextButton =
                (Button) getActivity()
                .findViewById(R.id.launch_next_activity_button);
    }
}
```

Validate Launch of Another Activity

Your unit testing goals might include:

- Verifying that `LaunchActivity` fires an `Intent` when a button is pushed clicked.
- Verifying that the launched `Intent` contains the correct payload data.

To verify if an `Intent` was triggered following the `Button` click, you can use the `getStartedActivityIntent()` method. By using assertion methods, you can verify that the returned `Intent` is not null, and that it contains the expected string value to launch the next `Activity`. If both assertions evaluate to `true`, you've successfully verified that the `Intent` was correctly sent by your `Activity`.

You might implement your test method like this:

```
@MediumTest
public void testNextActivityWasLaunchedWithIntent() {
    startActivity(mLaunchIntent, null, null);
    final Button launchNextButton =
            (Button) getActivity()
            .findViewById(R.id.launch_next_activity_button);
    launchNextButton.performClick();

    final Intent launchIntent = getStartedActivityIntent();
    assertNotNull("Intent was null", launchIntent);
    assertTrue(isFinishCalled());

    final String payload =

launchIntent.getStringExtra(NextActivity.EXTRAS_PAYLOAD_KEY);
    assertEquals("Payload is empty", LaunchActivity.STRING_PAYLOAD,
payload);
}
```

Because `LaunchActivity` runs in isolation, you cannot use the `TouchUtils` library to manipulate UI controls. To directly click a `Button`, you can call the `performClick()` method instead.

Next Get started Next class

Creating Functional Tests

Previous Next Get started

This lesson teaches you to

1. Add Test Method to Validate Functional Behavior
    1. Set Up an ActivityMonitor
    2. Send Keyboard Input Using Instrumentation

Try it out

Download the demo

AndroidTestingFun.zip

Functional testing involves verifying that individual application components work together as expected by the user. For example, you can create a functional test to verify that an `Activity` correctly launches a target `Activity` when the user performs a UI interaction.

To create a functional test for your `Activity`, your test class should extend `ActivityInstrumentationTestCase2`. Unlike `ActivityUnitTestCase`, tests in `ActivityInstrumentationTestCase2` can communicate with the Android system and send keyboard input and click events to the UI.

For a complete test case example, take a look at `SenderActivityTest.java` in the sample app.

Add Test Method to Validate Functional Behavior

Your functional testing goals might include:

- Verifying that a target `Activity` is started when a UI control is pushed in the sender `Activity`.
- Verifying that the target `Activity` displays the correct data based on the user's input in the sender `Activity`.

You might implement your test method like this:

```
@MediumTest
public void testSendMessageToReceiverActivity() {
    final Button sendToReceiverButton = (Button)
            mSenderActivity.findViewById(R.id.send_message_button);

    final EditText senderMessageEditText = (EditText)

mSenderActivity.findViewById(R.id.message_input_edit_text);

    // Set up an ActivityMonitor
    ...

    // Send string input value
    ...

    // Validate that ReceiverActivity is started
    ...

    // Validate that ReceiverActivity has the correct data
    ...

    // Remove the ActivityMonitor
    ...
}
```

The test waits for an `Activity` that matches this monitor, otherwise returns null after a timeout elapses. If `ReceiverActivity` was started, the `ActivityMonitor` that you set up earlier receives a hit. You can use the assertion methods to verify that the `ReceiverActivity` is indeed started, and that the hit count on the `ActivityMonitor` incremented as expected.

Set up an ActivityMonitor

To monitor a single `Activity` in your application, you can register an `ActivityMonitor`. The `ActivityMonitor` is notified by the system whenever an `Activity` that matches your criteria is started. If a match is found, the monitor’s hit count is updated.

Generally, to use an `ActivityMonitor`, you should:

1. Retrieve the `Instrumentation` instance for your test case by using the `getInstrumentation()` method.
2. Add an instance of `Instrumentation.ActivityMonitor` to the current instrumentation using one of the `Instrumentation` `addMonitor()` methods. The match criteria can be specified as an `IntentFilter` or a class name string.
3. Wait for the `Activity` to start.
4. Verify that the monitor hits were incremented.
5. Remove the monitor.

For example:

```
// Set up an ActivityMonitor
ActivityMonitor receiverActivityMonitor =

getInstrumentation().addMonitor(ReceiverActivity.class.getName(),
        null, false);

// Validate that ReceiverActivity is started
TouchUtils.clickView(this, sendToReceiverButton);
ReceiverActivity receiverActivity = (ReceiverActivity)

receiverActivityMonitor.waitForActivityWithTimeout(TIMEOUT_IN_MS);
assertNotNull("ReceiverActivity is null", receiverActivity);
assertEquals("Monitor for ReceiverActivity has not been called",
        1, receiverActivityMonitor.getHits());
assertEquals("Activity is of wrong type",
        ReceiverActivity.class, receiverActivity.getClass());

// Remove the ActivityMonitor
getInstrumentation().removeMonitor(receiverActivityMonitor);
```

Send Keyboard Input Using Instrumentation

If your `Activity` has an `EditText` field, you might want to test that users can enter values into the `EditText` object.

Generally, to send a string input value to an `EditText` object in `ActivityInstrumentationTestCase2`, you should:

1. Use the `runOnMainSync()` method to run the `requestFocus()` call synchronously in a loop. This way, the UI thread is blocked until focus is received.

2. Call `waitForIdleSync()` method to wait for the main thread to become idle (that is, have no more events to process).
3. Send a text string to the `EditText` by calling `sendStringSync()` and pass your input string as the parameter.

For example:

```
// Send string input value
getInstrumentation().runOnMainSync(new Runnable() {
    @Override
    public void run() {
        senderMessageEditText.requestFocus();
    }
});
getInstrumentation().waitForIdleSync();
getInstrumentation().sendStringSync("Hello Android!");
getInstrumentation().waitForIdleSync();
```

Next Get started Next class

Automating User Interface Tests

Previous Next Get started

You should also read

- Testing Support Library

User interface (UI) testing lets you ensure that your app meets its functional requirements and achieves a high standard of quality such that it is more likely to be successfully adopted by users.

One approach to UI testing is to simply have a human tester perform a set of user operations on the target app and verify that it is behaving correctly. However, this manual approach can be time-consuming, tedious, and error-prone. A more efficient approach is to write your UI tests such that user actions are performed in an automated way. The automated approach allows you to run your tests quickly and reliably in a repeatable manner.

Note: It is strongly encouraged that you use Android Studio for building your test apps, because it provides project setup, library inclusion, and packaging conveniences. This class assumes you are using Android Studio.

To automate UI tests with Android Studio, you implement your test code in a separate Android test folder (`src/androidTest/java`). The Android Plug-in for Gradle builds a test app based on your test code, then loads the test app on the same device as the target app. In your test code, you can use UI testing frameworks to simulate user interactions on the target app, in order to perform testing tasks that cover specific usage scenarios.

For testing Android apps, you typically create these types of automated UI tests:

- *UI tests that span a single app:* This type of test verifies that the target app behaves as expected when a user performs a specific action or enters a specific input in its activities. It allows you to check that the target app returns the correct UI output in response to user interactions in the app's activities. UI testing frameworks like Espresso allow you to programmatically simulate user actions and test complex intra-app user interactions.
- *UI tests that span multiple apps:* This type of test verifies the correct behavior of interactions between different user apps or between user apps and system apps. For example, you might want to test that your camera app shares images correctly with a 3rd-party social media app, or with the default Android Photos app. UI testing frameworks that support cross-app interactions, such as UI Automator, allow you to create tests for such scenarios.

The lessons in this class teach you how to use the tools and APIs in the Android Testing Support Library to build these types of automated tests. Before you begin building tests using these APIs, you must install the Android Testing Support Library, as described in Downloading the Android Testing Support Library.

Lessons

Testing UI for a Single App

Learn how to test UI in a single app by using the Espresso testing framework.

Testing UI for Multiple Apps

Learn how to test UI in multiple apps by using the UI Automator testing framework.

Next Get started Next class

Testing UI for a Single App

Previous Next Get started

Dependencies and Prerequisites

- Android 2.2 (API level 8) or higher
- Android Testing Support Library

This lesson teaches you to

1. Set Up Espresso
2. Create an Espresso Test Class
3. Run Espresso Tests on a Device or Emulator

You should also read

- Espresso API Reference

Try it out

- Espresso Code Samples

Testing user interactions within a single app helps to ensure that users do not encounter unexpected results or have a poor experience when interacting with your app. You should get into the habit of creating user interface (UI) tests if you need to verify that the UI of your app is functioning correctly.

The Espresso testing framework, provided by the Android Testing Support Library, provides APIs for writing UI tests to simulate user interactions within a single target app. Espresso tests can run on devices running

Android 2.2 (API level 8) and higher. A key benefit of using Espresso is that it provides automatic synchronization of test actions with the UI of the app you are testing. Espresso detects when the main thread is idle, so it is able to run your test commands at the appropriate time, improving the reliability of your tests. This capability also relieves you from having to adding any timing workarounds, such as a sleep period, in your test code.

The Espresso testing framework is an instrumentation-based API and works with the `AndroidJUnitRunner` test runner.

Set Up Espresso

Before you begin using Espresso, you must:

- **Install the Android Testing Support Library**. The Espresso API is located under the `com.android.support.test.espresso` package. These classes allow you to create tests that use the Espresso testing framework. To learn how to install the library, see Testing Support Library Setup.
- **Set up your project structure.** In your Gradle project, the source code for the target app that you want to test is typically placed under the `app/src/main` folder. The source code for instrumentation tests, including your Espresso tests, must be placed under the `app/src/androidTest` folder. To learn more about setting up your project directory, see Managing Projects.
- **Specify your Android testing dependencies**. In order for the Android Plug-in for Gradle to correctly build and run your Espresso tests, you must specify the following libraries in the `build.gradle` file of your Android app module:

  ```
  dependencies {
      androidTestCompile 'com.android.support.test:runner:0.3'
      androidTestCompile 'com.android.support.test:rules:0.3'
      androidTestCompile
  'com.android.support.test.espresso:espresso-core:2.2'
  }
  ```

- **Turn off animations on your test device.** Leaving system animations turned on in the test device might cause unexpected

results or may lead your test to fail. Turn off animations from *Settings* by opening *Developing Options* and turning all the following options off:
- *Window animation scale*
- *Transition animation scale*
- *Animator duration scale*

Create an Espresso Test Class

To create an Espresso test, create a Java class or an `ActivityInstrumentationTestCase2` subclass that follows this programming model:

1. Find the UI component you want to test in an `Activity` (for example, a sign-in button in the app) by calling the `onView()` method, or the `onData()` method for `AdapterView` controls.
2. Simulate a specific user interaction to perform on that UI component, by calling the `ViewInteraction.perform()` or `DataInteraction.perform()` method and passing in the user action (for example, click on the sign-in button). To sequence multiple actions on the same UI component, chain them using a comma-separated list in your method argument.
3. Repeat the steps above as necessary, to simulate a user flow across multiple activities in the target app.
4. Use the `ViewAssertions` methods to check that the UI reflects the expected state or behavior, after these user interactions are performed.

These steps are covered in more detail in the sections below.

The following code snippet shows how your test class might invoke this basic workflow:

```
onView(withId(R.id.my_view))            // withId(R.id.my_view) is a
ViewMatcher
        .perform(click())               // click() is a ViewAction
        .check(matches(isDisplayed())); // matches(isDisplayed()) is
a ViewAssertion
```

Using Espresso with ActivityInstrumentationTestCase2

If you are subclassing `ActivityInstrumentationTestCase2` to create your Espresso test class, you must inject an `Instrumentation` instance into your test class. This step is required in order for your Espresso test to run with the `AndroidJUnitRunner` test runner.

To do this, call the `injectInstrumentation()` method and pass in the result of `InstrumentationRegistry.getInstrumentation()`, as shown in the following code example:

```
import android.support.test.InstrumentationRegistry;

public class MyEspressoTest
        extends ActivityInstrumentationTestCase2<MyActivity> {

    private MyActivity mActivity;

    public MyEspressoTest() {
        super(MyActivity.class);
    }

    @Before
    public void setUp() throws Exception {
        super.setUp();

injectInstrumentation(InstrumentationRegistry.getInstrumentation());
        mActivity = getActivity();
    }

   ...
}
```

Note: Previously, `InstrumentationTestRunner` would inject the `Instrumentation` instance, but this test runner is being deprecated.

Accessing UI Components

Before Espresso can interact with the app under test, you must first specify the UI component or *view*. Espresso supports the use of Hamcrest matchers for specifying views and adapters in your app.

To find the view, call the `onView()` method and pass in a view matcher that specifies the view that you are targeting. This is described in more detail in Specifying a View Matcher. The `onView()` method returns a `ViewInteraction` object that allows your test to interact with the view.

However, calling the `onView()` method may not work if you want to locate a view in an `AdapterView` layout. In this case, follow the instructions in Locating a view in an AdapterView instead.

Note: The `onView()` method does not check if the view you specified is valid. Instead, Espresso searches only the current view hierarchy, using the matcher provided. If no match is found, the method throws a `NoMatchingViewException`.

The following code snippet shows how you might write a test that accesses an `EditText` field, enters a string of text, closes the virtual keyboard, and then performs a button click.

```
public void testChangeText_sameActivity() {
    // Type text and then press the button.
    onView(withId(R.id.editTextUserInput))
            .perform(typeText(STRING_TO_BE_TYPED),
closeSoftKeyboard());
    onView(withId(R.id.changeTextButton)).perform(click());

    // Check that the text was changed.
    ...
}
```

Specifying a View Matcher

You can specify a view matcher by using these approaches:

- Calling methods in the `ViewMatchers` class. For example, to find a view by looking for a text string it displays, you can call a method like this:

  ```
  onView(withText("Sign-in"));
  ```

  Similarly you can call `withId()` and providing the resource ID (`R.id`) of the view, as shown in the following example:

  ```
  onView(withId(R.id.button_signin));
  ```

  Android resource IDs are not guaranteed to be unique. If your test attempts to match to a resource ID used by more than one view, Espresso throws an `AmbiguousViewMatcherException`.

- Using the Hamcrest `Matchers` class. You can use the `allOf()`

methods to combine multiple matchers, such as `containsString()` and `instanceOf()`. This approach allows you to filter the match results more narrowly, as shown in the following example:

```
onView(allOf(withId(R.id.button_signin), withText("Sign-in")));
```

You can use the `not` keyword to filter for views that don't correspond to the matcher, as shown in the following example:

```
onView(allOf(withId(R.id.button_signin), not(withText("Sign-out"))));
```

To use these methods in your test, import the `org.hamcrest.Matchers` package. To learn more about Hamcrest matching, see the Hamcrest site.

To improve the performance of your Espresso tests, specify the minimum matching information needed to find your target view. For example, if a view is uniquely identifiable by its descriptive text, you do not need to specify that it is also assignable from the `TextView` instance.

Locating a view in an AdapterView

In an `AdapterView` widget, the view is dynamically populated with child views at runtime. If the target view you want to test is inside an `AdapterView` (such as a `ListView`, `GridView`, or `Spinner`), the `onView()` method might not work because only a subset of the views may be loaded in the current view hierarchy.

Instead, call the `onData()` method to obtain a `DataInteraction` object to access the target view element. Espresso handles loading the target view element into the current view hierarchy. Espresso also takes care of scrolling to the target element, and putting the element into focus.

Note: The `onData()` method does not check if if the item you specified corresponds with a view. Espresso searches only the current view hierarchy. If no match is found, the method throws a `NoMatchingViewException`.

The following code snippet shows how you can use the `onData()` method

together with Hamcrest matching to search for a specific row in a list that contains a given string. In this example, the `LongListActivity` class contains a list of strings exposed through a `SimpleAdapter`.

```
onData(allOf(is(instanceOf(Map.class)),
        hasEntry(equalTo(LongListActivity.ROW_TEXT), is(str))));
```

Performing Actions

Call the `ViewInteraction.perform()` or `DataInteraction.perform()` methods to simulate user interactions on the UI component. You must pass in one or more `ViewAction` objects as arguments. Espresso fires each action in sequence according to the given order, and executes them in the main thread.

The `ViewActions` class provides a list of helper methods for specifying common actions. You can use these methods as convenient shortcuts instead of creating and configuring individual `ViewAction` objects. You can specify such actions as:

- `ViewActions.click()`: Clicks on the view.
- `ViewActions.typeText()`: Clicks on a view and enters a specified string.
- `ViewActions.scrollTo()`: Scrolls to the view. The target view must be subclassed from `ScrollView` and the value of its `android:visibility` property must be `VISIBLE`. For views that extend `AdapterView` (for example, `ListView`), the `onData()` method takes care of scrolling for you.
- `ViewActions.pressKey()`: Performs a key press using a specified keycode.
- `ViewActions.clearText()`: Clears the text in the target view.

If the target view is inside a `ScrollView`, perform the `ViewActions.scrollTo()` action first to display the view in the screen before other proceeding with other actions. The `ViewActions.scrollTo()` action will have no effect if the view is already displayed.

Verifying Results

Call the `ViewInteraction.check()` or `DataInteraction.check()` method to assert that the view in the UI matches some expected state. You must pass in a `ViewAssertion` object as the argument. If the assertion fails, Espresso throws an `AssertionFailedError`.

The `ViewAssertions` class provides a list of helper methods for specifying common assertions. The assertions you can use include:

- `doesNotExist`: Asserts that there is no view matching the specified criteria in the current view hierarchy.
- `matches`: Asserts that the specified view exists in the current view hierarchy and its state matches some given Hamcrest matcher.
- `selectedDescendentsMatch` : Asserts that the specified children views for a parent view exist, and their state matches some given Hamcrest matcher.

The following code snippet shows how you might check that the text displayed in the UI has the same value as the text previously entered in the `EditText` field.

```
public void testChangeText_sameActivity() {
    // Type text and then press the button.
    ...

    // Check that the text was changed.
    onView(withId(R.id.textToBeChanged))
            .check(matches(withText(STRING_TO_BE_TYPED)));
}
```

Run Espresso Tests on a Device or Emulator

To run Espresso tests, you must use the `AndroidJUnitRunner` class provided in the Android Testing Support Library as your default test runner. The Android Plug-in for Gradle provides a default directory (`src/androidTest/java`) for you to store the instrumented test classes and test suites that you want to run on a device. The plug-in compiles the test code in that directory and then executes the test app using the configured test runner class.

To run Espresso tests in your Gradle project:

1. Specify `AndroidJUnitRunner` as the default test instrumentation runner in your `build.gradle` file:

   ```
   android {
       defaultConfig {
           testInstrumentationRunner
   "android.support.test.runner.AndroidJUnitRunner"
       }
   }
   ```

2. Run your tests from the command-line by calling the the `connectedCheck` (or `cC`) task:

   ```
   ./gradlew cC
   ```

Next Get started Next class

Testing UI for Multiple Apps

Previous Next Get started

Dependencies and Prerequisites

- Android 4.3 (API level 18) or higher
- Android Testing Support Library

This lesson teaches you to

1. Set Up UI Automator
2. Create a UI Automator Test Class
3. Run UI Automator Tests on a Device or Emulator

You should also read

- UI Automator API Reference

Try it out

- UI Automator Code Samples

A user interface (UI) test that involves user interactions across multiple apps lets you verify that your app behaves correctly when the user flow crosses into other apps or into the system UI. An example of such a user flow is a messaging app that lets the user enter a text message, launches the Android contact picker so that the users can select recipients to send the message to, and then returns control to the original app for the user to submit the message.

This lesson covers how to write such UI tests using the UI Automator

testing framework provided by the Android Testing Support Library. The UI Automator APIs let you interact with visible elements on a device, regardless of which `Activity` is in focus. Your test can look up a UI component by using convenient descriptors such as the text displayed in that component or its content description. UI Automator tests can run on devices running Android 4.3 (API level 18) or higher.

The UI Automator testing framework is an instrumentation-based API and works with the `AndroidJUnitRunner` test runner.

Set Up UI Automator

Before you begin using UI Automator, you must:

- **Install the Android Testing Support Library**. The UI Automator API is located under the `com.android.support.test.uiautomator` package. These classes allow you to create tests that use the Espresso testing framework. To learn how to install the library, see Testing Support Library Setup.
- **Set up your project structure.** In your Gradle project, the source code for the target app that you want to test is typically placed under the `app/src/main` folder. The source code for instrumentation tests, including your UI Automator tests, must be placed under the `app/src/androidTest` folder. To learn more about setting up your project directory, see Managing Projects.
- **Specify your Android testing dependencies**. In order for the Android Plug-in for Gradle to correctly build and run your UI Automator tests, you must specify the following libraries in the `build.gradle` file of your Android app module:

```
dependencies {
    androidTestCompile 'com.android.support.test:runner:0.3'
    androidTestCompile 'com.android.support.test:rules:0.3'
    androidTestCompile
'com.android.support.test.uiautomator:uiautomator-v18:2.1.1'
}
```

To optimize your UI Automator testing, you should first inspect the target app's UI components and ensure that they are accessible. These optimization tips are described in the next two sections.

Inspecting the UI on a device

Before designing your test, inspect the UI components that are visible on the device. To ensure that your UI Automator tests can access these components, check that these components have visible text labels, `android:contentDescription` values, or both.

The `uiautomatorviewer` tool provides a convenient visual interface to inspect the layout hierarchy and view the properties of UI components that are visible on the foreground of the device. This information lets you create more fine-grained tests using UI Automator. For example, you can create a UI selector that matches a specific visible property.

To launch the `uiautomatorviewer` tool:

1. Launch the target app on a physical device.
2. Connect the device to your development machine.
3. Open a terminal window and navigate to the `<android-sdk>/tools/` directory.
4. Run the tool with this command:

   ```
   $ uiautomatorviewer
   ```

To view the UI properties for your application:

1. In the `uiautomatorviewer` interface, click the **Device Screenshot** button.
2. Hover over the snapshot in the left-hand panel to see the UI components identified by the `uiautomatorviewertool`. The properties are listed in the lower right-hand panel and the layout hierarchy in the upper right-hand panel.
3. Optionally, click on the **Toggle NAF Nodes** button to see UI components that are non-accessible to UI Automator. Only limited information may be available for these components.

To learn about the common types of UI components provided by Android, see User Interface.

Ensuring your Activity is accessible

The UI Automator test framework depends on the accessibility features of the Android framework to look up individual UI elements. As a developer, you should implement these minimum optimizations in your `Activity` to support UI Automator:

- Use the `android:contentDescription` attribute to label the `ImageButton`, `ImageView`, `CheckBox` and other user interface controls.
- Provide an `android:hint` attribute instead of a content description for `EditText` fields.
- Associate an `android:hint` attribute with any graphical icons used by controls that provide feedback to the user (for example, status or state information).
- Use the `uiautomatorviewer` tool to ensure that the UI component is accessible to the testing framework. You can also test the application by turning on accessibility services like TalkBack and Explore by Touch, and try using your application using only directional controls.

Generally, app developers get accessibility support for free, courtesy of the `View` and `ViewGroup` classes. However, some apps use custom view elements to provide a richer user experience. Such custom elements won't get the accessibility support that is provided by the standard Android UI elements. If this applies to your app, make sure that it exposes the custom-drawn UI element to Android accessibility services by implementing the `AccessibilityNodeProvider` class.

If the custom view element contains a single element, make it accessible by implementing accessibility API methods. If the custom view contains elements that are not views themselves (for example, a `WebView`, make sure it implements the `AccessibilityNodeProvider` class. For container views that extend an existing container implementation (for example, a `ListView`), implementing `AccessibilityNodeProvider` is not necessary.

For more information about implementing and testing accessibility, see Making Applications Accessible.

Create a UI Automator Test Class

To build a UI Automator test, create a class that extends `InstrumentationTestCase`. Implement the following programming model in your UI Automator test class:

1. Get a `UiDevice` object to access the device you want to test, by calling the `getInstance()` method and passing it an `Instrumentation` object as the argument.
2. Get a `UiObject` object to access a UI component that is displayed on the device (for example, the current view in the foreground), by calling the `findObject()` method.
3. Simulate a specific user interaction to perform on that UI component, by calling a `UiObject` method; for example, call `performMultiPointerGesture()` to simulate a multi-touch gesture, and `setText()` to edit a text field. You can call on the APIs in steps 2 and 3 repeatedly as necessary to test more complex user interactions that involve multiple UI components or sequences of user actions.
4. Check that the UI reflects the expected state or behavior, after these user interactions are performed.

These steps are covered in more detail in the sections below.

Accessing UI Components

The `UiDevice` object is the primary way you access and manipulate the state of the device. In your tests, you can call `UiDevice` methods to check for the state of various properties, such as current orientation or display size. Your test can use the `UiDevice` object to perform device-level actions, such as forcing the device into a specific rotation, pressing D-pad hardware buttons, and pressing the Home and Menu buttons.

It's good practice to start your test from the Home screen of the device. From the Home screen (or some other starting location you've chosen in the device), you can call the methods provided by the UI Automator API to select and interact with specific UI elements.

The following code snippet shows how your test might get an instance of UiDevice and simulate a Home button press:

```
import android.test.InstrumentationTestCase;
import android.support.test.uiautomator.UiDevice;
import android.support.test.uiautomator.By;

public class CalculatorUiTest extends InstrumentationTestCase {

    private UiDevice mDevice;

    public void setUp() {
        // Initialize UiDevice instance
        mDevice = UiDevice.getInstance(getInstrumentation());

        // Start from the home screen
        mDevice.pressHome();

mDevice.wait(Until.hasObject(By.pkg(getHomeScreenPackage()).depth(0)),

    }
}
```

Use the findObject() method to retrieve a UiObject which represents a view that matches a given selector criteria. You can reuse the UiObject instances that you have created in other parts of your app testing, as needed. Note that the UI Automator test framework searches the current display for a match every time your test uses a UiObject instance to click on a UI element or query a property.

The following snippet shows how your test might construct UiObject instances that represent a Cancel button and a OK button in an app.

```
UiObject cancelButton = mDevice.findObject(new UiSelector()
        .text("Cancel"))
        .className("android.widget.Button"));
UiObject okButton = mDevice.findObject(new UiSelector()
        .text("OK"))
        .className("android.widget.Button"));

// Simulate a user-click on the OK button, if found.
if(okButton.exists() && okButton.isEnabled()) {
    okButton.click();
}
```

Specifying a selector

If you want to access a specific UI component in an app, use the UiSelector class. This class represents a query for specific elements in

the currently displayed UI.

If more than one matching element is found, the first matching element in the layout hierarchy is returned as the target `UiObject`. When constructing a `UiSelector`, you can chain together multiple properties to refine your search. If no matching UI element is found, a `UiAutomatorObjectNotFoundException` is thrown.

You can use the `childSelector()` method to nest multiple `UiSelector` instances. For example, the following code example shows how your test might specify a search to find the first `ListView` in the currently displayed UI, then search within that `ListView` to find a UI element with the text property Apps.

```
UiObject appItem = new UiObject(new UiSelector()
        .className("android.widget.ListView")
        .instance(1)
        .childSelector(new UiSelector()
        .text("Apps")));
```

As a best practice, when specifying a selector, you should use a Resource ID (if one is assigned to a UI element) instead of a text element or content-descriptor. Not all elements have a text element (for example, icons in a toolbar). Text selectors are brittle and can lead to test failures if there are minor changes to the UI. They may also not scale across different languages; your text selectors may not match translated strings.

It can be useful to specify the object state in your selector criteria. For example, if you want to select a list of all checked elements so that you can uncheck them, call the `checked()` method with the argument set to `true`.

Performing Actions

Once your test has obtained a `UiObject` object, you can call the methods in the `UiObject` class to perform user interactions on the UI component represented by that object. You can specify such actions as:

- `click()` : Clicks the center of the visible bounds of the UI element.
- `dragTo()` : Drags this object to arbitrary coordinates.

- `setText()` : Sets the text in an editable field, after clearing the field's content. Conversely, the `clearTextField()` method clears the existing text in an editable field.
- `swipeUp()` : Performs the swipe up action on the `UiObject`. Similarly, the `swipeDown()`, `swipeLeft()`, and `swipeRight()` methods perform corresponding actions.

The UI Automator testing framework allows you to send an `Intent` or launch an `Activity` without using shell commands, by getting a `Context` object through `getContext()`.

The following snippet shows how your test can use an `Intent` to launch the app under test. This approach is useful when you are only interested in testing the calculator app, and don't care about the launcher.

```
public void setUp() {
    ...

    // Launch a simple calculator app
    Context context = getInstrumentation().getContext();
    Intent intent = context.getPackageManager()
            .getLaunchIntentForPackage(CALC_PACKAGE);
    intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TASK);
            // Clear out any previous instances
    context.startActivity(intent);
    mDevice.wait(Until.hasObject(By.pkg(CALC_PACKAGE).depth(0)),
TIMEOUT);
}
```

Performing actions on collections

Use the `UiCollection` class if you want to simulate user interactions on a collection of items (for example, songs in a music album or a list of emails in an Inbox). To create a `UiCollection` object, specify a `UiSelector` that searches for a UI container or a wrapper of other child UI elements, such as a layout view that contains child UI elements.

The following code snippet shows how your test might construct a `UiCollection` to represent a video album that is displayed within a `FrameLayout`:

```
UiCollection videos = new UiCollection(new UiSelector()
        .className("android.widget.FrameLayout"));
```

```
// Retrieve the number of videos in this collection:
int count = videos.getChildCount(new UiSelector()
        .className("android.widget.LinearLayout"));

// Find a specific video and simulate a user-click on it
UiObject video = videos.getChildByText(new UiSelector()
        .className("android.widget.LinearLayout"), "Cute Baby
Laughing");
video.click();

// Simulate selecting a checkbox that is associated with the video
UiObject checkBox = video.getChild(new UiSelector()
        .className("android.widget.Checkbox"));
if(!checkBox.isSelected()) checkbox.click();
```

Performing actions on scrollable views

Use the `UiScrollable` class to simulate vertical or horizontal scrolling across a display. This technique is helpful when a UI element is positioned off-screen and you need to scroll to bring it into view.

The following code snippet shows how to simulate scrolling down the Settings menu and clicking on an About tablet option:

```
UiScrollable settingsItem = new UiScrollable(new UiSelector()
        .className("android.widget.ListView"));
UiObject about = settingsItem.getChildByText(new UiSelector()
        .className("android.widget.LinearLayout"), "About tablet");
about.click();
```

Verifying Results

The `InstrumentationTestCase` extends `TestCase`, so you can use standard JUnit `Assert` methods to test that UI components in the app return the expected results.

The following snippet shows how your test can locate several buttons in a calculator app, click on them in order, then verify that the correct result is displayed.

```
private static final String CALC_PACKAGE = "com.myexample.calc";

public void testTwoPlusThreeEqualsFive() {
    // Enter an equation: 2 + 3 = ?
    mDevice.findObject(new UiSelector()
            .packageName(CALC_PACKAGE).resourceId("two")).click();
    mDevice.findObject(new UiSelector()
            .packageName(CALC_PACKAGE).resourceId("plus")).click();
    mDevice.findObject(new UiSelector()
```

```
            .packageName(CALC_PACKAGE).resourceId("three")).click();
    mDevice.findObject(new UiSelector()
            .packageName(CALC_PACKAGE).resourceId("equals")).click();

    // Verify the result = 5
    UiObject result = mDevice.findObject(By.res(CALC_PACKAGE,
"result"));
    assertEquals("5", result.getText());
}
```

Run UI Automator Tests on a Device or Emulator

UI Automator tests are based on the `Instrumentation` class. The Android Plug-in for Gradle provides a default directory (`src/androidTest/java`) for you to store the instrumented test classes and test suites that you want to run on a device. The plug-in compiles the test code in that directory and then executes the test app using a test runner class. You are strongly encouraged to use the `AndroidJUnitRunner` class provided in the Android Testing Support Library as your default test runner.

To run UI Automator tests in your Gradle project:

1. Specify `AndroidJUnitRunner` as the default test instrumentation runner in your `build.gradle` file:

   ```
   android {
       defaultConfig {
           testInstrumentationRunner
   "android.support.test.runner.AndroidJUnitRunner"
       }
   }
   ```

2. Run your tests from the command-line by calling the `connectedCheck` (or `cC`) task:

   ```
   ./gradlew cC
   ```

Next Get started Next class

Building Effective Unit Tests

Previous Next Get started

You should also read

- Testing Support Library

Unit tests are the fundamental tests in your app testing strategy. By creating and running unit tests against your code, you can easily verify that the logic of individual units is correct. Running unit tests after every build helps you to quickly catch and fix software regressions introduced by code changes to your app.

A unit test generally exercises the functionality of the smallest possible unit of code (which could be a method, class, or component) in a repeatable way. You should build unit tests when you need to verify the logic of specific code in your app. For example, if you are unit testing a class, your test might check that the class is in the right state. Typically, the unit of code is tested in isolation; your test affects and monitors changes to that unit only. A mocking framework can be used to isolate your unit from its dependencies.

Note: Unit tests are not suitable for testing complex UI interaction events. Instead, you should use the UI testing frameworks, as described in Automating UI Tests.

For testing Android apps, you typically create these types of automated unit tests:

- **Local tests:** Unit tests that run on your local machine only. These tests are compiled to run locally on the Java Virtual Machine (JVM) to minimize execution time. Use this approach to run unit tests that have no dependencies on the Android framework or have

dependencies that can be filled by using mock objects.

- **Instrumented tests:** Unit tests that run on an Android device or emulator. These tests have access to instrumentation information, such as the `Context` for the app under test. Use this approach to run unit tests that have Android dependencies which cannot be easily filled by using mock objects.

The lessons in this class teach you how to build these types of automated unit tests.

Lessons

Building Local Unit Tests
Learn how to build unit tests that run on your local machine.

Building Instrumented Unit Tests
Learn how to build unit tests that run on an Android device or emulator.

Next Get started Next class

Building Local Unit Tests

Previous Next Get started

Dependencies and Prerequisites

- Android Plug-in for Gradle 1.1.0 or higher

This lesson teaches you to

1. Set Up Your Testing Environment
2. Create a Local Unit Test Class
3. Run Local Unit Tests

Try it out

- Local Unit Tests Code Samples

If your unit test has no dependencies or only has simple dependencies on Android, you should run your test on a local development machine. This testing approach is efficient because it helps you avoid the overhead of loading the target app and unit test code onto a physical device or emulator every time your test is run. Consequently, the execution time for running your unit test is greatly reduced. With this approach, you normally use a mocking framework, like Mockito, to fulfill any dependency relationships.

Android Plug-in for Gradle version 1.1.0 and higher allows you to create a source directory (`src/test/java`) in your project to store JUnit tests that you want to run on a local machine. This feature improves your project organization by letting you group your unit tests together into a single source set. You can run the tests from Android Studio or the command-line, and the plugin executes them on the local Java Virtual Machine

(JVM) on your development machine.

Set Up Your Testing Environment

Before building local unit tests, you must:

- **Set up your project structure.** In your Gradle project, the source code for the target app that you want to test is typically placed under the `app/src/main/java` folder. The source code for your local unit tests must be placed under the `app/src/test/java` folder. To learn more about setting up your project directory, see Run Local Unit Tests and Managing Projects.
- **Specify your Android testing dependencies**. In order to use JUnit 4 and Mockito with your local unit tests, specify the following libraries in the `build.gradle` file of your Android app module:

```
dependencies {
    // Unit testing dependencies
    testCompile 'junit:junit:4.12'
    // Set this dependency if you want to use Mockito
    testCompile 'org.mockito:mockito-core:1.10.19'
    // Set this dependency if you want to use Hamcrest matching
    androidTestCompile 'org.hamcrest:hamcrest-library:1.1'
}
```

Create a Local Unit Test Class

Your local unit test class should be written as a JUnit 4 test class. JUnit is the most popular and widely-used unit testing framework for Java. The latest version of this framework, JUnit 4, allows you to write tests in a cleaner and more flexible way than its predecessor versions. Unlike the previous approach to Android unit testing based on JUnit 3, with JUnit 4, you do not need to extend the `junit.framework.TestCase` class. You also do not need to prefix your test method name with the `'test'` keyword, or use any classes in the `junit.framework` or `junit.extensions` package.

To create a basic JUnit 4 test class, create a Java class that contains one or more test methods. A test method begins with the `@Test` annotation and contains the code to exercise and verify a single functionality in the component that you want to test.

The following example shows how you might implement a local unit test class. The test method `emailValidator_CorrectEmailSimple_ReturnsTrue` verifies that the `isValidEmail()` method in the app under test returns the correct result.

```
import org.junit.Test;
import java.util.regex.Pattern;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;

public class EmailValidatorTest {

    @Test
    public void emailValidator_CorrectEmailSimple_ReturnsTrue() {
        assertThat(EmailValidator.isValidEmail("name@email.com"),
is(true));
    }
    ...
}
```

To test that components in your app return the expected results, use the junit.Assert methods to perform validation checks (or *assertions*) to compare the state of the component under test against some expected value. To make tests more readable, you can use Hamcrest matchers (such as the `is()` and `equalTo()` methods) to match the returned result against the expected result.

In your JUnit 4 test class, you can use annotations to call out sections in your test code for special processing, such as:

- `@Before`: Use this annotation to specify a block of code with test setup operations. This code block will be invoked before each test. You can have multiple `@Before` methods but the order which these methods are called is not fixed.
- `@After`: This annotation specifies a block of code with test tear-down operations. This code block will be called after every test method. You can define multiple `@After` operations in your test code. Use this annotation to release any resources from memory.
- `@Test`: Use this annotation to mark a test method. A single test class can contain multiple test methods, each prefixed with this annotation.
- `@BeforeClass`: Use this annotation to specify static methods to be invoked only once per test class. This testing step is useful for

expensive operations such as connecting to a database.

- `@AfterClass`: Use this annotation to specify static methods to be invoked only after all tests in the class have been run. This testing step is useful for releasing any resources allocated in the `@BeforeClass` block.
- `@Test(timeout=<milliseconds>)`: Specifies a timeout period for the test. If the test starts but does not complete within the given timeout period, it automatically fails. You must specify the timeout period in milliseconds, for example: `@Test(timeout=5000)`.

Mocking Android dependencies

By default, the Android Plug-in for Gradle executes your local unit tests against a modified version of the `android.jar` library, which does not contain any actual code. Instead, method calls to Android classes from your unit test throw an exception.

You can use a mocking framework to stub out external dependencies in your code, to easily test that your component interacts with a dependency in an expected way. By substituting Android dependencies with mock objects, you can isolate your unit test from the rest of the Android system while verifying that the correct methods in those dependencies are called. The Mockito mocking framework for Java (version 1.9.5 and higher) offers compatibility with Android unit testing. With Mockito, you can configure mock objects to return some specific value when invoked.

To add a mock object to your local unit test using this framework, follow this programming model:

1. Include the Mockito library dependency in your `build.gradle` file, as described in Set Up Your Testing Environment.
2. At the beginning of your unit test class definition, add the `@RunWith(MockitoJUnitRunner.class)` annotation. This annotation tells the Mockito test runner to validate that your usage of the framework is correct and simplifies the initialization of your mock objects.
3. To create a mock object for an Android dependency, add the `@Mock` annotation before the field declaration.

4. To stub the behavior of the dependency, you can specify a condition and return value when the condition is met by using the `when()` and `thenReturn()` methods.

The following example shows how you might create a unit test that uses a mock `Context` object.

```
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.CoreMatchers.*;
import static org.mockito.Mockito.*;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.Mock;
import org.mockito.runners.MockitoJUnitRunner;
import android.content.SharedPreferences;

@RunWith(MockitoJUnitRunner.class)
public class UnitTestSample {

    private static final String FAKE_STRING = "HELLO WORLD";

    @Mock
    Context mMockContext;

    @Test
    public void readStringFromContext_LocalizedString() {
        // Given a mocked Context injected into the object under
test...
        when(mMockContext.getString(R.string.hello_word))
                .thenReturn(FAKE_STRING);
        ClassUnderTest myObjectUnderTest = new
ClassUnderTest(mMockContext);

        // ...when the string is returned from the object under
test...
        String result = myObjectUnderTest.getHelloWorldString();

        // ...then the result should be the expected one.
        assertThat(result, is(FAKE_STRING));
    }
}
```

To learn more about using the Mockito framework, see the Mockito API reference and the `SharedPreferencesHelperTest` class in the sample code.

Run Local Unit Tests

The Android Plug-in for Gradle provides a default directory (`src/test/java`) for you to store unit test classes that you want to run on a local JVM. The plug-in compiles the test code in that directory and then

executes the test app locally using the default test runner class.

As with production code, you can create unit tests for a specific flavor or build type. You should keep unit tests in a test source tree location that corresponds to your production source tree, such as:

Path to Production Class	Path to Local Unit Test Class
`src/main/java/Foo.java`	`src/test/java/FooTest.java`
`src/debug/java/Foo.java`	`src/testDebug/java/FooTest.java`
`src/myFlavor/java/Foo.java`	`src/testMyFlavor/java/FooTest.java`

Running local unit tests from Android Studio

To run local unit tests in your Gradle project from Android Studio:

1. In the **Project** window, right click on the project and synchronize your project.
2. Open the **Build Variants** window by clicking the left-hand tab, then change the test artifact to *Unit Tests*.
3. In the **Project** window, drill down to your unit test class or method, then right-click and run it.

Android Studio displays the results of the unit test execution in the **Run** window.

Running local unit tests from the command-line

To run local unit tests in your Gradle project from the command-line, call the `test` task command with the `--continue` option.

```
./gradlew test --continue
```

If there are failing tests, the command will display links to HTML reports (one per build variant). You can find the generated HTML test result reports in the `<path_to_your_project>/app/build/reports/tests/` directory, and the corresponding XML files in the `<path_to_your_project>/app/build/test-results/` directory.

Next Get started Next class

Building Instrumented Unit Tests

Previous Next Get started

Dependencies and Prerequisites

- Android 2.2 (API level 8) or higher
- Android Testing Support Library

This lesson teaches you to

1. Set Up Your Testing Environment
2. Create a Instrumented Unit Test Class
3. Run Instrumented Unit Tests

Try it out

- Instrumented Unit Tests Code Samples

Instrumented unit tests are unit tests that run on physical devices and emulators, instead of the Java Virtual Machine (JVM) on your local machine. You should create instrumented unit tests if your tests need access to instrumentation information (such as the target app's `Context`) or if they require the real implementation of an Android framework component (such as a `Parcelable` or `SharedPreferences` object). Using instrumented unit tests also helps to reduce the effort required to write and maintain mock code. You are still free to use a mocking framework, if you choose, to simulate any dependency relationships. Instrumented unit tests can take advantage of the Android framework APIs and supporting APIs, such as the Android Testing Support Library.

Set Up Your Testing Environment

Before building instrumented unit tests, you must:

- **Install the Android Testing Support Library**. The `AndroidJUnitRunner` API, located under the `com.android.support.test.runner` package, allows you to create and run instrumented unit tests. To learn how to install the library, see Testing Support Library Setup.
- **Set up your project structure.** In your Gradle project, the source code for the target app that you want to test is typically placed under the `app/src/main/java` folder. The source code for instrumentatation tests, including your unit tests, must be placed under the `app/src/androidTest/java` folder. To learn more about setting up your project directory, see Managing Projects.
- **Specify your Android testing dependencies**. In order for the Android Plug-in for Gradle to correctly build and run your instrumented unit tests, you must specify the following libraries in the `build.gradle` file of your Android app module:

```
dependencies {
    androidTestCompile 'com.android.support.test:runner:0.3'
    androidTestCompile 'com.android.support.test:rules:0.3'
    // Set this dependency if you want to use Hamcrest matching
    androidTestCompile 'org.hamcrest:hamcrest-library:1.1'
}
```

Create an Instrumented Unit Test Class

Your instrumented unit test class should be written as a JUnit 4 test class. To learn more about creating JUnit 4 test classes and using JUnit 4 assertions and annotations, see Create a Local Unit Test Class.

To create an instrumented JUnit 4 test class, add the `@RunWith(AndroidJUnit4.class)` annotation at the beginning of your test class definition. You also need to specify the `AndroidJUnitRunner` class provided in the Android Testing Support Library as your default test runner. This step is described in more detail in Run Instrumented Unit Tests.

The following example shows how you might write an instrumented unit test to test that the `Parcelable` interface is implemented correctly for the `LogHistory` class:

```
import android.os.Parcel;
import android.support.test.runner.AndroidJUnit4;
import android.util.Pair;
import org.junit.Test;
import org.junit.runner.RunWith;
import java.util.List;
import static org.hamcrest.Matchers.is;
import static org.junit.Assert.assertThat;

@RunWith(AndroidJUnit4.class)
public class LogHistoryAndroidUnitTest {

    public static final String TEST_STRING = "This is a string";
    public static final long TEST_LONG = 12345678L;
    private LogHistory mLogHistory;

    @Before
    public void createLogHistory() {
        mLogHistory = new LogHistory();
    }

    @Test
    public void logHistory_ParcelableWriteRead() {
        // Set up the Parcelable object to send and receive.
        mLogHistory.addEntry(TEST_STRING, TEST_LONG);

        // Write the data.
        Parcel parcel = Parcel.obtain();
        mLogHistory.writeToParcel(parcel,
mLogHistory.describeContents());

        // After you're done with writing, you need to reset the
parcel for reading.
        parcel.setDataPosition(0);

        // Read the data.
        LogHistory createdFromParcel =
LogHistory.CREATOR.createFromParcel(parcel);
        List<Pair<String, Long>> createdFromParcelData =
createdFromParcel.getData();

        // Verify that the received data is correct.
        assertThat(createdFromParcelData.size(), is(1));
        assertThat(createdFromParcelData.get(0).first,
is(TEST_STRING));
        assertThat(createdFromParcelData.get(0).second, is(TEST_LONG));
    }
}
```

Creating a test suite

To organize the execution of your instrumented unit tests, you can group a collection of test classes in a *test suite* class and run these tests together. Test suites can be nested; your test suite can group other test suites and run all their component test classes together.

A test suite is contained in a test package, similar to the main application

package. By convention, the test suite package name usually ends with the `.suite` suffix (for example, `com.example.android.testing.mysample.suite`).

To create a test suite for your unit tests, import the JUnit `RunWith` and `Suite` classes. In your test suite, add the `@RunWith(Suite.class)` and the `@Suite.SuitClasses()` annotations. In the `@Suite.SuiteClasses()` annotation, list the individual test classes or test suites as arguments.

The following example shows how you might implement a test suite called `UnitTestSuite` that groups and runs the `CalculatorInstrumentationTest` and `CalculatorAddParameterizedTest` test classes together.

```
import
com.example.android.testing.mysample.CalculatorAddParameterizedTest;
import
com.example.android.testing.mysample.CalculatorInstrumentationTest;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

// Runs all unit tests.
@RunWith(Suite.class)
@Suite.SuiteClasses({CalculatorInstrumentationTest.class,
        CalculatorAddParameterizedTest.class})
public class UnitTestSuite {}
```

Run Instrumented Unit Tests

The Android Plug-in for Gradle provides a default directory (`src/androidTest/java`) for you to store the instrumented unit and integration test classes and test suites that you want to run on a device. The plug-in compiles the test code in that directory and then executes the test app using a test runner class. You must set the `AndroidJUnitRunner` class provided in the Android Testing Support Library as your default test runner.

To specify `AndroidJUnitRunner` as the default test instrumentation runner, add the following setting in your `build.gradle` file:

```
android {
    defaultConfig {
        testInstrumentationRunner
"android.support.test.runner.AndroidJUnitRunner"
    }
}
```

Running instrumented unit tests from Android Studio

To run instrumented unit tests in your Gradle project from Android Studio:

1. Open the **Build Variants** window by clicking the left-hand tab, then set the test artifact to *Android Instrumentation Tests*.
2. In the **Project** window, drill down to your unit test class or method, then right-click and run it using the Android Test configuration.

Android Studio displays the results of the unit test execution in the **Run** window.

Running instrumented unit tests from the command-line

To run instrumented unit tests in your Gradle project from the command-line, call the `connectedCheck` (or `cC`) task:

```
./gradlew cC
```

You can find the generated HTML test result reports in the `<path_to_your_project>/app/build/outputs/reports/androidTests/connected/` directory, and the corresponding XML files in the `<path_to_your_project>/app/build/outputs/androidTest-results/connected/` directory.

Next Get started Next class

Using Google Play to Distribute & Monetize

These classes focus on the business aspects of your app strategy, including techniques for distributing your app on Google Play and techniques for building revenue.

Selling In-app Products

Previous Next Get started

Dependencies and prerequisites

- Android 2.2 or higher

You Should Also Read

- In-app Billing Overview

In this class, you'll learn how to perform common In-app Billing operations from Android applications.

In-app billing is a service hosted on Google Play that lets you charge for digital content or for upgrades in your app. The In-app Billing API makes it easy for you to integrate In-app Billing into your applications. You can request product details from Google Play, issue orders for in-app products, and quickly retrieve ownership information based on users' purchase history. You can also query the Google Play service for details about in-app products, such as local pricing and availability. Google Play provides a checkout interface that makes user interactions with the In-app Billing service seamless, and provides a more intuitive experience to your users.

This class describes how to get started with the Version 3 API. To learn how to use the version 2 API, see Implementing In-App Billing (V2).

Lessons

Preparing Your In-app Billing Application

In this lesson, you will learn how to prepare your application to use

the In-app Billing API and communicate with Google Play. You will also learn how to establish a connection to communicate with Google Play and verify that the In-app Billing API version that you are using is supported.

Establishing In-app Billing Products for Sale

In this lesson, you will learn how to specify the In-app Billing products for your app in Google Play and query the product details.

Purchase In-app Billing Products

In this lesson, you will learn how to purchase In-app Billing products, track consumption of purchased items, and query for details of purchased items.

Testing Your In-app Billing Application

In this lesson, you will learn how to test your application to ensure that In-app Billing is functioning correctly.

Next Get started Next class

Preparing Your In-app Billing Application

Previous Next Get started

This lesson teaches you to

1. Download the Sample App
2. Add Your App to the Developer Console
3. Add the In-app Billing Library
4. Set the Billing Permission
5. Initiate a Connection with Google Play

You should also read

- In-app Billing Overview

Before you can start using the In-app Billing service, you'll need to add the library that contains the In-app Billing Version 3 API to your Android project. You also need to set the permissions for your application to communicate with Google Play. In addition, you'll need to establish a connection between your application and Google Play. You should also verify that the In-app Billing API version that you are using in your application is supported by Google Play.

Download the Sample Application

In this training class, you will use a reference implementation for the In-app Billing Version 3 API called the `TrivialDrive` sample application. The sample includes convenience classes to quickly set up the In-app Billing service, marshal and unmarshal data types, and handle In-app Billing requests from the main thread of your application.

To download the sample application:

1. Open the Android SDK Manager.
2. In the SDK Manager, expand the `Extras` section.
3. Select **Google Play Billing Library**.
4. Click **Install packages** to complete the download.

The sample files will be installed to `<sdk>/extras/google/play_billing/`.

Add Your Application to the Developer Console

The Google Play Developer Console is where you publish your In-app Billing application and manage the various digital goods that are available for purchase from your application. When you create a new application entry in the Developer Console, it automatically generates a public license key for your application. You will need this key to establish a trusted connection from your application to the Google Play servers. You only need to generate this key once per application, and don't need to repeat these steps when you update the APK file for your application.

To add your application to the Developer Console:

1. Go to the Google Play Developer Console site and log in. You will need to register for a new developer account, if you have not registered previously. To sell in-app items, you also need to have a Google payments merchant account.
2. Click on **Try the new design** to access the preview version of the Developer Console, if you are not already logged on to that version.
3. In the **All Applications** tab, add a new application entry.
   a. Click **Add new application**.
   b. Enter a name for your new In-app Billing application.
   c. Click **Prepare Store Listing**.
4. In the **Services & APIs** tab, find and make a note of the public license key that Google Play generated for your application. This is a Base64 string that you will need to include in your application code

later.

Your application should now appear in the list of applications in Developer Console.

Add the In-app Billing Library

To use the In-app Billing Version 3 features, you must add the `IInAppBillingService.aidl` file to your Android project. This Android Interface Definition Language (AIDL) file defines the interface to the Google Play service.

You can find the `IInAppBillingService.aidl` file in the provided sample app. Depending on whether you are creating a new application or modifying an existing application, follow the instructions below to add the In-app Billing Library to your project.

New Project

To add the In-app Billing Version 3 library to your new In-app Billing project:

1. Copy the `TrivialDrive` sample files into your Android project.
2. Modify the package name in the files you copied to use the package name for your project. In Eclipse, you can use this shortcut: right-click the package name, then select **Refactor** > **Rename**.
3. Open the `AndroidManifest.xml` file and update the package attribute value to use the package name for your project.
4. Fix import statements as needed so that your project compiles correctly. In Eclipse, you can use this shortcut: press **Ctrl+Shift+O** in each file showing errors.
5. Modify the sample to create your own application. Remember to copy the Base64 public license key for your application from the Developer Console over to your `MainActivity.java`.

Existing Project

To add the In-app Billing Version 3 library to your existing In-app Billing project:

1. Copy the `IInAppBillingService.aidl` file to your Android project.
   - In Android Studio: Create a directory named `aidl` under `src/main`, add a new package `com.android.vending.billing` in this directory, and import the `IInAppBillingService.aidl` file into this package.
   - In Eclipse: Import the `IInAppBillingService.aidl` file into your `/src` directory.
   - In other dev environments: Create the following directory `/src/com/android/vending/billing` and copy the `IInAppBillingService.aidl` file into this directory.
2. Build your application. You should see a generated file named `IInAppBillingService.java` in the `/gen` directory of your project.
3. Add the helper classes from the `/util` directory of the `TrivialDrive` sample to your project. Remember to change the package name declarations in those files accordingly so that your project compiles correctly.

Your project should now contain the In-app Billing Version 3 library.

Set the Billing Permission

Your app needs to have permission to communicate request and response messages to the Google Play's billing service. To give your app the necessary permission, add this line in your `AndroidManifest.xml` manifest file:

```
<uses-permission android:name="com.android.vending.BILLING" />
```

Initiate a Connection with Google Play

You must bind your Activity to Google Play's In-app Billing service to send In-app Billing requests to Google Play from your application. The convenience classes provided in the sample handles the binding to the In-app Billing service, so you don't have to manage the network

connection directly.

To set up synchronous communication with Google Play, create an `IabHelper` instance in your activity's `onCreate` method. In the constructor, pass in the `Context` for the activity, along with a string containing the public license key that was generated earlier by the Google Play Developer Console.

Security Recommendation: It is highly recommended that you do not hard-code the exact public license key string value as provided by Google Play. Instead, you can construct the whole public license key string at runtime from substrings, or retrieve it from an encrypted store, before passing it to the constructor. This approach makes it more difficult for malicious third-parties to modify the public license key string in your APK file.

```
IabHelper mHelper;

@Override
public void onCreate(Bundle savedInstanceState) {
   // ...
   String base64EncodedPublicKey;

   // compute your public key and store it in base64EncodedPublicKey
   mHelper = new IabHelper(this, base64EncodedPublicKey);
}
```

Next, perform the service binding by calling the `startSetup` method on the `IabHelper` instance that you created. Pass the method an `OnIabSetupFinishedListener` instance, which is called once the `IabHelper` completes the asynchronous setup operation. As part of the setup process, the `IabHelper` also checks if the In-app Billing Version 3 API is supported by Google Play. If the API version is not supported, or if an error occured while establishing the service binding, the listener is notified and passed an `IabResult` object with the error message.

```
mHelper.startSetup(new IabHelper.OnIabSetupFinishedListener() {
   public void onIabSetupFinished(IabResult result) {
      if (!result.isSuccess()) {
         // Oh noes, there was a problem.
         Log.d(TAG, "Problem setting up In-app Billing: " + result);
      }
         // Hooray, IAB is fully set up!
   }
});
```

If the setup completed successfully, you can now use the `mHelper` reference to communicate with the Google Play service. When your application is launched, it is a good practice to query Google Play to find out what in-app items are owned by a user. This is covered further in the Query Purchased Items section.

Important: Remember to unbind from the In-app Billing service when you are done with your activity. If you don't unbind, the open service connection could cause your device's performance to degrade. To unbind and free your system resources, call the `IabHelper`'s `dispose` method when your `Activity` is destroyed.

```
@Override
public void onDestroy() {
   super.onDestroy();
   if (mHelper != null) mHelper.dispose();
   mHelper = null;
}
```

Next Get started Next class

Establishing In-app Billing Products for Sale

Previous Next Get started

This lesson teaches you to

1. Specify In-app Products in Google Play
2. Query In-app Product Details

You should also read

- In-app Billing Overview

Before publishing your In-app Billing application, you'll need to define the product list of digital goods available for purchase in the Google Play Developer Console.

Specify In-app Products in Google Play

From the Developer Console, you can define product information for in-app products and associate the product list with your application.

To add new in-app products to your product list:

1. Build a signed APK file for your In-app Billing application. To learn how to build and sign your APK, see Building Your Application for Release. Make sure that you are using your final (not debug) certificate and private key to sign your application.
2. In the Developer Console, open the application entry that you created earlier.
3. Click on the APK tab then click on Upload new APK. Upload the

signed APK file to the Developer Console. Don't publish the app yet!

4. Navigate to the uploaded app listing, and click on **In-app Products**.
5. Click on the option to add a new product, then complete the form to specify the product information such as the item's unique product ID (also called its *SKU*), description, price, and country availability. Note down the product ID since you might need this information to query purchase details in your application later.

   Important: The In-app Billing Version 3 service only supports managed in-app products, so make sure that you specify that the purchase type is 'Managed' when you add new items to your product list in the Developer Console.

6. Once you have completed the form, activate the product so that your application can purchase it.

   Warning: It may take up to 2-3 hours after uploading the APK for Google Play to recognize your updated APK version. If you try to test your application before your uploaded APK is recognized by Google Play, your application will receive a 'purchase cancelled' response with an error message "This version of the application is not enabled for In-app Billing."

Query Items Available for Purchase

You can query Google Play to programmatically retrieve details of the in-app products that are associated with your application (such as the product's price, title, description, and type). This is useful, for example, when you want to display a listing of unowned items that are still available for purchase to users.

Note: When making the query, you will need to specify the product IDs for the products explicitly. You can manually find the product IDs from the Developer Console by opening the **In-app Products** tab for your application. The product IDs are listed under the column labeled **Name/ID**.

To retrieve the product details, call `queryInventoryAsync(boolean, List, QueryInventoryFinishedListener)` on your IabHelper instance.

- The first input argument indicates whether product details should be retrieved (should be set to `true`).
- The `List` argument consists of one or more product IDs (also called SKUs) for the products that you want to query.
- Finally, the `QueryInventoryFinishedListener` argument specifies a listener is notified when the query operation has completed and handles the query response.

If you use the convenience classes provided in the sample, the classes will handle background thread management for In-app Billing requests, so you can safely make queries from the main thread of your application.

The following code shows how you can retrieve the details for two products with IDs `SKU_APPLE` and `SKU_BANANA` that you previously defined in the Developer Console.

```
List additionalSkuList = new List();
additionalSkuList.add(SKU_APPLE);
additionalSkuList.add(SKU_BANANA);
mHelper.queryInventoryAsync(true, additionalSkuList,
   mQueryFinishedListener);
```

If the query is successful, the query results are stored in an `Inventory` object that is passed back to the listener.

The following code shows how you can retrieve the item prices from the result set.

```
IabHelper.QueryInventoryFinishedListener
   mQueryFinishedListener = new
IabHelper.QueryInventoryFinishedListener() {
   public void onQueryInventoryFinished(IabResult result, Inventory
inventory)
   {
      if (result.isFailure()) {
         // handle error
         return;
       }

       String applePrice =
          inventory.getSkuDetails(SKU_APPLE).getPrice();
       String bananaPrice =
          inventory.getSkuDetails(SKU_BANANA).getPrice();
```

```
        // update the UI
    }
}
```

Next Get started Next class

Purchasing In-app Billing Products

Previous Next Get started

This lesson teaches you to

1. Purchase an Item
2. Query Purchased Items
3. Consume a Purchase

You should also read

- In-app Billing Overview

Once your application is connected to Google Play, you can initiate purchase requests for in-app products. Google Play provides a checkout interface for users to enter their payment method, so your application does not need to handle payment transactions directly.

When an item is purchased, Google Play recognizes that the user has ownership of that item and prevents the user from purchasing another item with the same product ID until it is consumed. You can control how the item is consumed in your application, and notify Google Play to make the item available for purchase again.

You can also query Google Play to quickly retrieve the list of purchases that were made by the user. This is useful, for example, when you want to restore the user's purchases when your user launches your app.

Purchase an Item

To start a purchase request from your app, call `launchPurchaseFlow(Activity, String, int, OnIabPurchaseFinishedListener, String)` on your `IabHelper` instance. You must make this call from the main thread of your `Activity`. Here's an explaination of the `launchPurchaseFlow` method parameters:

- The first argument is the calling `Activity`.
- The second argument is the product ID (also called its SKU) of the item to purchase. Make sure that you are providing the ID and not the product name. You must have previously defined and activated the item in the Developer Console, otherwise it won't be recognized.
- The third argument is a request code value. This value can be any positive integer. Google Play reurns this request code to the calling `Activity`'s `onActivityResult` along with the purchase response.
- The fourth argument is a listener that is notified when the purchase operation has completed and handles the purchase response from Google Play.
- The fifth argument contains a 'developer payload' string that you can use to send supplemental information about an order (it can be an empty string). Typically, this is used to pass in a string token that uniquely identifies this purchase request. If you specify a string value, Google Play returns this string along with the purchase response. Subsequently, when you make queries about this purchase, Google Play returns this string together with the purchase details.

  Security Recommendation: It's good practice to pass in a string that helps your application to identify the user who made the purchase, so that you can later verify that this is a legitimate purchase by that user. For consumable items, you can use a randomly generated string, but for non-consumable items you should use a string that uniquely identifies the user.

The following example shows how you can make a purchase request for a product with ID `SKU_GAS`, using an arbitrary value of 10001 for the request code, and an encoded developer payload string.

```
mHelper.launchPurchaseFlow(this, SKU_GAS, 10001,
```

```
   mPurchaseFinishedListener,
"bGoa+V7g/yqDXvKRqq+JTFn4uQZbPiQJo4pf9RzJ");
```

If the purchase order is successful, the response data from Google Play is stored in an `Purchase` object that is passed back to the listener.

The following example shows how you can handle the purchase response in the listener, depending on whether the purchase order was completed successfully, and whether the user purchased gas or a premium upgrade. In this example, gas is an in-app product that can be purchased multiple times, so you should consume the purchase to allow the user to buy it again. To learn how to consume purchases, see the Consuming Products section. The premium upgrade is a one-time purchase so you don't need to consume it. It is good practice to update the UI immediately so that your users can see their newly purchased items.

```
IabHelper.OnIabPurchaseFinishedListener mPurchaseFinishedListener
   = new IabHelper.OnIabPurchaseFinishedListener() {
   public void onIabPurchaseFinished(IabResult result, Purchase
purchase)
   {
      if (result.isFailure()) {
         Log.d(TAG, "Error purchasing: " + result);
         return;
      }
      else if (purchase.getSku().equals(SKU_GAS)) {
         // consume the gas and update the UI
      }
      else if (purchase.getSku().equals(SKU_PREMIUM)) {
         // give user access to premium content and update the UI
      }
   }
};
```

Security Recommendation: When you receive the purchase response from Google Play, make sure to check the returned data signature, the `orderId`, and the `developerPayload` string in the `Purchase` object to make sure that you are getting the expected values. You should verify that the `orderId` is a unique value that you have not previously processed, and the `developerPayload` string matches the token that you sent previously with the purchase request. As a further security precaution, you should perform the verification on your own secure server.

Query Purchased Items

Upon a successful purchase, the user's purchase data is cached locally by Google Play's In-app Billing service. It is good practice to frequently query the In-app Billing service for the user's purchases, for example whenever the app starts up or resumes, so that the user's current in-app product ownership information is always reflected in your app.

To retrieve the user's purchases from your app, call `queryInventoryAsync(QueryInventoryFinishedListener)` on your `IabHelper` instance. The `QueryInventoryFinishedListener` argument specifies a listener that is notified when the query operation has completed and handles the query response. It is safe to make this call fom your main thread.

```
mHelper.queryInventoryAsync(mGotInventoryListener);
```

If the query is successful, the query results are stored in an `Inventory` object that is passed back to the listener. The In-app Billing service returns only the purchases made by the user account that is currently logged in to the device.

```
IabHelper.QueryInventoryFinishedListener mGotInventoryListener
   = new IabHelper.QueryInventoryFinishedListener() {
   public void onQueryInventoryFinished(IabResult result,
      Inventory inventory) {

      if (result.isFailure()) {
        // handle error here
      }
      else {
        // does the user have the premium upgrade?
        mIsPremium = inventory.hasPurchase(SKU_PREMIUM);
        // update UI accordingly
      }
   }
};
```

Consume a Purchase

You can use the In-app Billing Version 3 API to track the ownership of purchased items in Google Play. Once an item is purchased, it is considered to be "owned" and cannot be purchased again from Google Play while in that state. You must send a consumption request for the item before Google Play makes it available for purchase again. All managed in-app products are consumable. How you use the consumption

mechanism in your app is up to you. Typically, you would implement consumption for products with temporary benefits that users may want to purchase multiple times (for example, in-game currency or replenishable game tokens). You would typically not want to implement consumption for products that are purchased once and provide a permanent effect (for example, a premium upgrade).

It's your responsibility to control and track how the in-app product is provisioned to the user. For example, if the user purchased in-game currency, you should update the player's inventory with the amount of currency purchased.

Security Recommendation: You must send a consumption request before provisioning the benefit of the consumable in-app purchase to the user. Make sure that you have received a successful consumption response from Google Play before you provision the item.

To record a purchase consumption, call `consumeAsync(Purchase, OnConsumeFinishedListener)` on your `IabHelper` instance. The first argument that the method takes is the `Purchase` object representing the item to consume. The second argument is a `OnConsumeFinishedListener` that is notified when the consumption operation has completed and handles the consumption response from Google Play. It is safe to make this call fom your main thread.

In this example, you want to consume the gas item that the user has previously purchased in your app.

```
mHelper.consumeAsync(inventory.getPurchase(SKU_GAS),
   mConsumeFinishedListener);
```

The following example shows how to implement the `OnConsumeFinishedListener`.

```
IabHelper.OnConsumeFinishedListener mConsumeFinishedListener =
   new IabHelper.OnConsumeFinishedListener() {
   public void onConsumeFinished(Purchase purchase, IabResult
result) {
      if (result.isSuccess()) {
         // provision the in-app purchase to the user
         // (for example, credit 50 gold coins to player's
character)
      }
```

```
        else {
            // handle error
        }
    }
};
```

Check for Consumable Items on Startup

It's important to check for consumable items when the user starts up your application. Typically, you would first query the In-app Billing service for the items purchased by the user (via `queryInventoryAsync`), then get the consumable `Purchase` objects from the Inventory. If your application detects that are any consumable items that are owned by the user, you should send a consumption request to Google Play immediately and provision the item to the user. See the `TrivialDrive` sample for an example of how to implement this checking at startup.

Next Get started Next class

Testing Your In-app Billing Application

Previous Next Get started

This lesson teaches you to

1. Test with Static Responses
2. Test with Your Own Product IDs

You should also read

- Testing In-app Billing

To ensure that In-app Billing is functioning correctly in your application, you should test the test the application before you publish it on Google Play. Early testing also helps to ensure that the user flow for purchasing in-app items is not confusing or slow, and that users can see their newly purchased items in a timely way.

Test with Static Responses

Test your In-app Billing application with static responses by using Google Play's reserved product IDs.By using reserved product IDs instead of actual product IDs, you can test the purchase flow without specifying an actual payment method or transferring money. To learn more about the reserved product IDs, see Testing In-app Billing.

Test with Your Own Product IDs

Because Google Play does not allow you to use your developer account

to directly purchase in-app products that you have created yourself, you'll need to create test acccounts under your developer account profile. To create a test account, simply enter a valid Google email address. Users with these test accounts will then be able to make in-app-billing purchases from uploaded, unpublished applications that you manage.

To test your In-app Billing Version 3 application using your own product IDs:

1. In the Developer Console, add one or more tester accounts to the developer account that you are using to publish your application.
   a. Login to the Developer Console with your developer account.
   b. Click **Settings** > **Account** details, then in the **License Testing** section, add the Google email addresses for your tester accounts.
2. Build a signed APK file for your In-app Billing application. To learn how to build and sign your APK, see Building Your Application for Release. Make sure that you are using your final (not debug) certificate and private key to sign your application.
3. Make sure that you have uploaded the signed APK for your application to the Developer Console, and associated one or more in-app products with your application. You don't need to publish the application on Google Play to test it.

   Warning: It may take up to 2-3 hours after uploading the APK for Google Play to recognize your updated APK version. If you try to test your application before your uploaded APK is recognized by Google Play, your application will receive a 'purchase cancelled' response with an error message "This version of the application is not enabled for In-app Billing."

4. Install the APK file to your physical test device by using the `adb` tool. To learn how to install the application, see Running on a Device. Make sure that:
   - Your test device is running on Android SDK Version 2.2 (API level 8) or higher, and is installed with Google Play client Version 3.9.16 or higher.
   - The `android:versionCode` and `android:versionName` attributes

values in the `AndroidManifest.xml` of the application that you are installing matches the values of your APK in the Developer Console.
    - Your application is signed with the same certificate that you used for the APK that you uploaded to the Developer Console, before installing it on your device.
5. Login to the test device by using a tester account. Test your In-app Billing application by purchasing a few items, and fix any issues that you encounter. To learn more about how you can perform large-scale testing of your In-app Billing app, see Test Purchases (In-app Billing Sandbox).

Next Get started Next class

Maintaining Multiple APKs

Previous Next Get started

Dependencies and prerequisites

- Android 1.0 and higher
- You must have an Google Play publisher account

You should also read

- Multiple APK Support

Multiple APK support is a feature of Google Play that allows you to publish multiple APKs under the same application listing. Each APK is a complete instance of your application, optimized to target specific device configurations. Each APK can target a specific set of GL textures, API levels, screen sizes, or some combination thereof.

This class shows you how to write your multiple APK application using any one of these configuration variables. Each lesson covers basics about how to organize your codebase and target the right devices, as well as the smart way to avoid pitfalls such as unnecessary redundancy across your codebase, and making mistakes in your manifest that could render an APK invisible to all devices on Google Play. By going through any of these lessons, you'll know how to develop multiple APKs the smart way, make sure they're targeting the devices you want them to, and know how to catch mistakes *before* your app goes live.

Lessons

Creating Multiple APKs for Different API Levels

Learn how to target different versions of the Android platform using

multiple APKs. Also learn how to organize your codebase, what to do with your manifest, and how to investigate your APK configuration using the `aapt` tool before pushing live.

Creating Multiple APKs for Different Screen Sizes

Learn how to target Android devices by screen size using multiple APKs. Also learn how to organize your codebase, what to do with your manifest, and how to investigate your APK configuration using the `aapt` tool before pushing live.

Creating Multiple APKs for Different GL Textures

Learn how to target Android devices based on their support for GL texture. Also learn how to organize your codebase, what to do with your manifest, and how to investigate your APK configuration using the `aapt` tool before pushing live.

Creating Multiple APKs with 2+ Dimensions

Learn how to target different Android devices based on more than one configuration variable (screen size, API version, GL texture). Examples in the lesson target using a combination of API level and screen size. Also learn how to organize your codebase, what to do with your manifest, and how to investigate your APK configuration using the `aapt` tool before pushing live.

Next Get started Next class

Creating Multiple APKs for Different API Levels

Previous Next Get started

This lesson teaches you to

1. Confirm You Need Multiple APKs
2. Chart Your Requirements
3. Put All Common Code and Resources in a Library Project
4. Create New APK Projects
5. Adjust the Manifests
6. Go Over Pre-launch Checklist

You should also read

- Multiple APK Support
- How to have your (Cup)cake and eat it too

When developing your Android application to take advantage of multiple APKs on Google Play, it's important to adopt some good practices from the get-go, and prevent unnecessary headaches further into the development process. This lesson shows you how to create multiple APKs of your app, each covering a slightly different range of API levels. You will also gain some tools necessary to make maintaining a multiple APK codebase as painless as possible.

Confirm You Need Multiple APKs

When trying to create an application that works across multiple generations of the Android platform, naturally you want your application to take advantage of new features on new devices, without sacrificing

backwards compatibility. It may seem at the outset as though multiple APK support is the best solution, but this often isn't the case. The Using Single APK Instead section of the multiple APK developer guide includes some useful information on how to accomplish this with a single APK, including use of our support library. You can also learn how to write code that runs only at certain API levels in a single APK, without resorting to computationally expensive techniques like reflection from this article.

If you can manage it, confining your application to a single APK has several advantages, including:

- Publishing and testing are easier
- There's only one codebase to maintain
- Your application can adapt to device configuration changes
- App restore across devices just works
- You don't have to worry about market preference, behavior from "upgrades" from one APK to the next, or which APK goes with which class of devices

The rest of this lesson assumes that you've researched the topic, studiously absorbed the material in the resources linked, and determined that multiple APKs are the right path for your application.

Chart Your Requirements

Start off by creating a simple chart to quickly determine how many APKs you need, and what API range each APK covers. For handy reference, the Platform Versions page of the Android Developer website provides data about the relative number of active devices running a given version of the Android platform. Also, although it sounds easy at first, keeping track of which set of API levels each APK is going to target gets difficult rather quickly, especially if there's going to be some overlap (there often is). Fortunately, it's easy to chart out your requirements quickly, easily, and have an easy reference for later.

In order to create your multiple APK chart, start out with a row of cells representing the various API levels of the Android platform. Throw an

extra cell at the end to represent future versions of Android.

3	4	5	6	7	8	9	10	11	12	13	+

Now just color in the chart such that each color represents an APK. Here's one example of how you might apply each APK to a certain range of API levels.

3	4	5	6	7	8	9	10	11	12	13	+

Once you've created this chart, distribute it to your team. Team communication on your project just got immediately simpler, since instead of asking "How's the APK for API levels 3 to 6, er, you know, the Android 1.x one. How's that coming along?" You can simply say "How's the Blue APK coming along?"

Put All Common Code and Resources in a Library Project

Whether you're modifying an existing Android application or starting one from scratch, this is the first thing that you should do to the codebase, and by the far the most important. Everything that goes into the library project only needs to be updated once (think language-localized strings, color themes, bugs fixed in shared code), which improves your development time and reduces the likelihood of mistakes that could have been easily avoided.

Note: While the implementation details of how to create and include library projects are beyond the scope of this lesson, you can get up to speed quickly on their creation at the following links:

- Setting up a library project (Eclipse)
- Setting up a library project (Command line)

If you're converting an existing application to use multiple APK support,

scour your codebase for every localized string file, list of values, theme colors, menu icons and layout that isn't going to change across APKs, and put it all in the library project. Code that isn't going to change much should also go in the library project. You'll likely find yourself extending these classes to add a method or two from APK to APK.

If, on the other hand, you're creating the application from scratch, try as much as possible to write code in the library project *first*, then only move it down to an individual APK if necessary. This is much easier to manage in the long run than adding it to one, then another, then another, then months later trying to figure out whether this blob can be moved up to the library section without screwing anything up.

Create New APK Projects

There should be a separate Android project for each APK you're going to release. For easy organization, place the library project and all related APK projects under the same parent folder. Also remember that each APK needs to have the same package name, although they don't necessarily need to share the package name with the library. If you were to have 3 APKs following the scheme described earlier, your root directory might look like this:

```
alexlucas:~/code/multi-apks-root$ ls
foo-blue
foo-green
foo-lib
foo-red
```

Once the projects are created, add the library project as a reference to each APK project. If possible, define your starting Activity in the library project, and extend that Activity in your APK project. Having a starting activity defined in the library project gives you a chance to put all your application initialization in one place, so that each individual APK doesn't have to re-implement "universal" tasks like initializing Analytics, running licensing checks, and any other initialization procedures that don't change much from APK to APK.

Adjust the Manifests

When a user downloads an application which uses multiple APKs through Google Play, the correct APK to use is chosen using two simple rules:

- The manifest has to show that particular APK is eligible
- Of the eligible APKs, highest version number wins

By way of example, let's take the set of multiple APKs described earlier, and assume that we haven't set a max API level for any of the APKs. Taken individually, the possible range of each APK would look like this:

3	4	5	6	7	8	9	10	11	12	13	+
				7	8	9	10	11	12	13	+
								11	12	13	+

Because it is required that an APK with a higher minSdkVersion also have a higher version code, we know that in terms of versionCode values, red ≥ green ≥ blue. Therefore we can effectively collapse the chart to look like this:

3	4	5	6	7	8	9	10	11	12	13	+

Now, let's further assume that the Red APK has some requirement on it that the other two don't. Filters on Google Play page of the Android Developer guide has a whole list of possible culprits. For the sake of example, let's assume that red requires a front-facing camera. In fact, the entire point of the red APK is to combine the front-facing camera with sweet new functionality that was added in API 11. But, it turns out, not all devices that support API 11 even HAVE front-facing cameras! The horror!

Fortunately, if a user is browsing Google Play from one such device, Google Play will look at the manifest, see that Red lists the front-facing camera as a requirement, and quietly ignore it, having determined that

Red and that device are not a match made in digital heaven. It will then see that Green is not only forward-compatible with devices with API 11 (since no maxSdkVersion was defined), but also doesn't care whether or not there's a front-facing camera! The app can still be downloaded from Google Play by the user, because despite the whole front-camera mishap, there was still an APK that supported that particular API level.

In order to keep all your APKs on separate "tracks", it's important to have a good version code scheme. The recommended one can be found on the Version Codes area of our developer guide. Since the example set of APKs is only dealing with one of 3 possible dimensions, it would be sufficient to separate each APK by 1000, set the first couple digits to the minSdkVersion for that particular APK, and increment from there. This might look like:

Blue: 03001, 03002, 03003, 03004...
Green: 07001, 07002, 07003, 07004...
Red:11001, 11002, 11003, 11004...

Putting this all together, your Android Manifests would likely look something like the following:

Blue:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="03001" android:versionName="1.0"
package="com.example.foo">
    <uses-sdk android:minSdkVersion="3" />
    ...
```

Green:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="07001" android:versionName="1.0"
package="com.example.foo">
    <uses-sdk android:minSdkVersion="7" />
    ...
```

Red:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="11001" android:versionName="1.0"
package="com.example.foo">
    <uses-sdk android:minSdkVersion="11" />
    ...
```

Go Over Pre-launch Checklist

Before uploading to Google Play, double-check the following items. Remember that these are specifically relevant to multiple APKs, and in no way represent a complete checklist for all applications being uploaded to Google Play.

- All APKs must have the same package name
- All APKs must be signed with the same certificate
- If the APKs overlap in platform version, the one with the higher minSdkVersion must have a higher version code
- Double check your manifest filters for conflicting information (an APK that only supports cupcake on XLARGE screens isn't going to be seen by anybody)
- Each APK's manifest must be unique across at least one of supported screen, openGL texture, or platform version
- Try to test each APK on at least one device. Barring that, you have one of the most customizable device emulators in the business sitting on your development machine. Go nuts!

It's also worth inspecting the compiled APK before pushing to market, to make sure there aren't any surprises that could hide your application on Google Play. This is actually quite simple using the "aapt" tool. Aapt (the Android Asset Packaging Tool) is part of the build process for creating and packaging your Android applications, and is also a very handy tool for inspecting them.

```
>aapt dump badging
package: name='com.example.hello' versionCode='1' versionName='1.0'
sdkVersion:'11'
uses-permission:'android.permission.SEND_SMS'
application-label:'Hello'
application-icon-120:'res/drawable-ldpi/icon.png'
application-icon-160:'res/drawable-mdpi/icon.png'
application-icon-240:'res/drawable-hdpi/icon.png'
application: label='Hello' icon='res/drawable-mdpi/icon.png'
launchable-activity: name='com.example.hello.HelloActivity'
label='Hello' icon=''
uses-feature:'android.hardware.telephony'
uses-feature:'android.hardware.touchscreen'
main
supports-screens: 'small' 'normal' 'large' 'xlarge'
```

```
supports-any-density: 'true'
locales: '--_--'
densities: '120' '160' '240'
```

When you examine aapt output, be sure to check that you don't have conflicting values for supports-screens and compatible-screens, and that you don't have unintended "uses-feature" values that were added as a result of permissions you set in the manifest. In the example above, the APK won't be visible to very many devices.

Why? By adding the required permission SEND_SMS, the feature requirement of android.hardware.telephony was implicitly added. Since API 11 is Honeycomb (the version of Android optimized specifically for tablets), and no Honeycomb devices have telephony hardware in them, Google Play will filter out this APK in all cases, until future devices come along which are higher in API level AND possess telephony hardware.

Fortunately this is easily fixed by adding the following to your manifest:

```
<uses-feature android:name="android.hardware.telephony"
android:required="false" />
```

The `android.hardware.touchscreen` requirement is also implicitly added. If you want your APK to be visible on TVs which are non-touchscreen devices you should add the following to your manifest:

```
<uses-feature android:name="android.hardware.touchscreen"
android:required="false" />
```

Once you've completed the pre-launch checklist, upload your APKs to Google Play. It may take a bit for the application to show up when browsing Google Play, but when it does, perform one last check. Download the application onto any test devices you may have, to make sure that the APKs are targeting the intended devices. Congratulations, you're done!

Next Get started Next class

Creating Multiple APKs for Different Screen Sizes

Previous Next Get started

This lesson teaches you to

1. Confirm You Need Multiple APKs
2. Chart Your Requirements
3. Put All Common Code and Resources in a Library Project.
4. Create New APK Projects
5. Adjust the Manifests
6. Go Over Pre-launch Checklist

You should also read

- Multiple APK Support
- Supporting Multiple Screens

When developing your Android application to take advantage of multiple APKs on Google Play, it's important to adopt some good practices from the get-go, and prevent unnecessary headaches further into the development process. This lesson shows you how to create multiple APKs of your app, each covering a different class of screen size. You will also gain some tools necessary to make maintaining a multiple APK codebase as painless as possible.

Confirm You Need Multiple APKs

When trying to create an application that works across multiple sizes of Android devices, naturally you want your application to take advantage of all the available space on larger devices, without sacrificing compatibility

or usability on the smaller screens. It may seem at the outset as though multiple APK support is the best solution, but this often isn't the case. The Using Single APK Instead section of the multiple APK developer guide includes some useful information on how to accomplish this with a single APK, including use of our support library. You should also read the guide to supporting multiple screens, and there's even a support library you can download using the Android SDK, which lets you use fragments on pre-Honeycomb devices (making multiple-screen support in a single APK much easier).

If you can manage it, confining your application to a single APK has several advantages, including:

- Publishing and testing are easier
- There's only one codebase to maintain
- Your application can adapt to device configuration changes
- App restore across devices just works
- You don't have to worry about market preference, behavior from "upgrades" from one APK to the next, or which APK goes with which class of devices

The rest of this lesson assumes that you've researched the topic, studiously absorbed the material in the resources linked, and determined that multiple APKs are the right path for your application.

Chart Your Requirements

Start off by creating a simple chart to quickly determine how many APKs you need, and what screen size(s) each APK covers. Fortunately, it's easy to chart out your requirements quickly and easily, and have a reference for later. Start out with a row of cells representing the various screen sizes available on the Android platform.

small	normal	large	xlarge

Now just color in the chart such that each color represents an APK.

Here's one example of how you might apply each APK to a certain range of screen sizes.

small	normal	large	xlarge

Depending on your needs, you could also have two APKs, "small and everything else" or "xlarge and everything else". Coloring in the chart also makes intra-team communication easier—You can now simply refer to each APK as "blue", "green", or "red", no matter how many different screen types it covers.

Put All Common Code and Resources in a Library Project.

Whether you're modifying an existing Android application or starting one from scratch, this is the first thing that you should do to the codebase, and by the far the most important. Everything that goes into the library project only needs to be updated once (think language-localized strings, color themes, bugs fixed in shared code), which improves your development time and reduces the likelihood of mistakes that could have been easily avoided.

Note: While the implementation details of how to create and include library projects are beyond the scope of this lesson, you can get up to speed quickly on their creation at the following links:

- Setting up a library project (Eclipse)
- Setting up a library project (Command line)

If you're converting an existing application to use multiple APK support, scour your codebase for every localized string file, list of values, theme colors, menu icons and layout that isn't going to change across APKs, and put it all in the library project. Code that isn't going to change much should also go in the library project. You'll likely find yourself extending these classes to add a method or two from APK to APK.

If, on the other hand, you're creating the application from scratch, try as much as possible to write code in the library project *first*, then only move it down to an individual APK if necessary. This is much easier to manage in the long run than adding it to one, then another, then another, then months later trying to figure out whether this blob can be moved up to the library section without screwing anything up.

Create New APK Projects

There should be a separate Android project for each APK you're going to release. For easy organization, place the library project and all related APK projects under the same parent folder. Also remember that each APK needs to have the same package name, although they don't necessarily need to share the package name with the library. If you were to have 3 APKs following the scheme described earlier, your root directory might look like this:

```
alexlucas:~/code/multi-apks-root$ ls
foo-blue
foo-green
foo-lib
foo-red
```

Once the projects are created, add the library project as a reference to each APK project. If possible, define your starting Activity in the library project, and extend that Activity in your APK project. Having a starting activity defined in the library project gives you a chance to put all your application initialization in one place, so that each individual APK doesn't have to re-implement "universal" tasks like initializing Analytics, running licensing checks, and any other initialization procedures that don't change much from APK to APK.

Adjust the Manifests

When a user downloads an application which uses multiple APKs through Google Play, the correct APK to use is chosen using two simple rules:

- The manifest has to show that particular APK is eligible
- Of the eligible APKs, highest version number wins

By way of example, let's take the set of multiple APKs described earlier, and assume that each APK has been set to support all screen sizes larger than its "target" screen size. Taken individually, the possible range of each APK would look like this:

small	normal	large	xlarge
		large	xlarge
			xlarge

However, by using the "highest version number wins" rule, if we set the versionCode attribute in each APK such that red ≥ green ≥ blue, the chart effectively collapses down to this:

small	normal	large	xlarge

Now, let's further assume that the Red APK has some requirement on it that the other two don't. The Filters on Google Play page of the Android Developer guide has a whole list of possible culprits. For the sake of example, let's assume that red requires a front-facing camera. In fact, the entire point of the red APK is to use the extra available screen space to do entertaining things with that front-facing camera. But, it turns out, not all xlarge devices even HAVE front-facing cameras! The horror!

Fortunately, if a user is browsing Google Play from one such device, Google Play will look at the manifest, see that Red lists the front-facing camera as a requirement, and quietly ignore it, having determined that Red and that device are not a match made in digital heaven. It will then see that Green is not only compatible with xlarge devices, but also doesn't care whether or not there's a front-facing camera! The app can still be downloaded from Google Play by the user, because despite the whole front-camera mishap, there was still an APK that supported that particular screen size.

In order to keep all your APKs on separate "tracks", it's important to have a good version code scheme. The recommended one can be found on the Version Codes area of our developer guide. Since the example set of APKs is only dealing with one of 3 possible dimensions, it would be sufficient to separate each APK by 1000 and increment from there. This might look like:

Blue: 1001, 1002, 1003, 1004...
Green: 2001, 2002, 2003, 2004...
Red:3001, 3002, 3003, 3004...

Putting this all together, your Android Manifests would likely look something like the following:

Blue:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="1001" android:versionName="1.0"
package="com.example.foo">
    <supports-screens android:smallScreens="true"
        android:normalScreens="true"
        android:largeScreens="true"
        android:xlargeScreens="true" />
    ...
```

Green:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="2001" android:versionName="1.0"
package="com.example.foo">
    <supports-screens android:smallScreens="false"
        android:normalScreens="false"
        android:largeScreens="true"
        android:xlargeScreens="true" />
    ...
```

Red:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="3001" android:versionName="1.0"
package="com.example.foo">
    <supports-screens android:smallScreens="false"
        android:normalScreens="false"
        android:largeScreens="false"
        android:xlargeScreens="true" />
    ...
```

Note that technically, multiple APK's will work with either the supports-

screens tag, or the compatible-screens tag. Supports-screens is generally preferred, and it's generally a really bad idea to use both tags in the same manifest. It makes things needlessly complicated, and increases the opportunity for errors. Also note that instead of taking advantage of the default values (small and normal are always true by default), the manifests explicitly set the value for each screen size. This can save you headaches down the line. For instance, a manifest with a target SDK of < 9 will have xlarge automatically set to false, since that size didn't exist yet. So be explicit!

Go Over Pre-launch Checklist

Before uploading to Google Play, double-check the following items. Remember that these are specifically relevant to multiple APKs, and in no way represent a complete checklist for all applications being uploaded to Google Play.

- All APKs must have the same package name
- All APKs must be signed with the same certificate
- Every screen size you want your APK to support, set to true in the manifest. Every screen size you want it to avoid, set to false
- Double check your manifest filters for conflicting information (an APK that only supports cupcake on XLARGE screens isn't going to be seen by anybody)
- Each APK's manifest must be unique across at least one of supported screen, openGL texture, or platform version
- Try to test each APK on at least one device. Barring that, you have one of the most customizable device emulators in the business sitting on your development machine. Go nuts!

It's also worth inspecting the compiled APK before pushing to market, to make sure there aren't any surprises that could hide your application on Google Play. This is actually quite simple using the "aapt" tool. Aapt (the Android Asset Packaging Tool) is part of the build process for creating and packaging your Android applications, and is also a very handy tool for inspecting them.

```
>aapt dump badging
package: name='com.example.hello' versionCode='1' versionName='1.0'
sdkVersion:'11'
uses-permission:'android.permission.SEND_SMS'
application-label:'Hello'
application-icon-120:'res/drawable-ldpi/icon.png'
application-icon-160:'res/drawable-mdpi/icon.png'
application-icon-240:'res/drawable-hdpi/icon.png'
application: label='Hello' icon='res/drawable-mdpi/icon.png'
launchable-activity: name='com.example.hello.HelloActivity'
label='Hello' icon=''
uses-feature:'android.hardware.telephony'
uses-feature:'android.hardware.touchscreen'
main
supports-screens: 'xlarge'
supports-any-density: 'true'
locales: '--_--'
densities: '120' '160' '240'
```

When you examine aapt output, be sure to check that you don't have conflicting values for supports-screens and compatible-screens, and that you don't have unintended "uses-feature" values that were added as a result of permissions you set in the manifest. In the example above, the APK will be invisible to most, if not all devices.

Why? By adding the required permission SEND_SMS, the feature requirement of android.hardware.telephony was implicitly added. Since most (if not all) xlarge devices are tablets without telephony hardware in them, Google Play will filter out this APK in these cases, until future devices come along which are both large enough to report as xlarge screen size, and possess telephony hardware.

Fortunately this is easily fixed by adding the following to your manifest:

```
<uses-feature android:name="android.hardware.telephony"
android:required="false" />
```

The `android.hardware.touchscreen` requirement is also implicitly added. If you want your APK to be visible on TVs which are non-touchscreen devices you should add the following to your manifest:

```
<uses-feature android:name="android.hardware.touchscreen"
android:required="false" />
```

Once you've completed the pre-launch checklist, upload your APKs to Google Play. It may take a bit for the application to show up when browsing Google Play, but when it does, perform one last check.

Download the application onto any test devices you may have to make sure that the APKs are targeting the intended devices. Congratulations, you're done!

Next Get started Next class

Creating Multiple APKs for Different GL Textures

Previous Next Get started

This lesson teaches you to

1. Confirm You Need Multiple APKs
2. Chart Your Requirements
3. Put All Common Code and Resources in a Library Project
4. Create New APK Projects
5. Adjust the Manifests
6. Go Over Pre-launch Checklist

You should also read

- Multiple APK Support

When developing your Android application to take advantage of multiple APKs on Google Play, it's important to adopt some good practices from the get-go, and prevent unnecessary headaches further into the development process. This lesson shows you how to create multiple APKs of your app, each supporting a different subset of OpenGL texture formats. You will also gain some tools necessary to make maintaining a multiple APK codebase as painless as possible.

Confirm You Need Multiple APKs

When trying to create an application that works across all available Android-powered devices, naturally you want your application look its best on each individual device, regardless of the fact they don't all support the same set of GL textures. It may seem at the outset as

though multiple APK support is the best solution, but this often isn't the case. The Using Single APK Instead section of the multiple APK developer guide includes some useful information on how to accomplish this with a single APK, including how to detect supported texture formats at runtime. Depending on your situation, it might be easier to bundle all formats with your application, and simply pick which one to use at runtime.

If you can manage it, confining your application to a single APK has several advantages, including:

- Publishing and Testing are easier
- There's only one codebase to maintain
- Your application can adapt to device configuration changes
- App restore across devices just works
- You don't have to worry about market preference, behavior from "upgrades" from one APK to the next, or which APK goes with which class of devices

The rest of this lesson assumes that you've researched the topic, studiously absorbed the material in the resources linked, and determined that multiple APKs are the right path for your application.

Chart Your Requirements

The Android Developer Guide provides a handy reference of some of common supported textures on the supports-gl-texture page. This page also contains some hints as to which phones (or families of phones) support particular texture formats. Note that it's generally a good idea for one of your APKs to support ETC1, as that texture format is supported by all Android-powered devices that support the OpenGL ES 2.0 spec.

Since most Android-powered devices support more than one texture format, you need to establish an order of preference. Create a chart including all the formats that your application is going to support. The left-most cell is going to be the lowest priority (It will probably be ETC1, a really solid default in terms of performance and compatibility). Then color in the chart such that each cell represents an APK.

ETC1	ATI	PowerVR

Coloring in the chart does more than just make this guide less monochromatic - It also has a way of making intra-team communication easier- You can now simply refer to each APK as "blue", "green", or "red", instead of "The one that supports ETC1 texture formats", etc.

Put All Common Code and Resources in a Library Project

Whether you're modifying an existing Android application or starting one from scratch, this is the first thing that you should do to the codebase, and by the far the most important. Everything that goes into the library project only needs to be updated once (think language-localized strings, color themes, bugs fixed in shared code), which improves your development time and reduces the likelihood of mistakes that could have been easily avoided.

Note: While the implementation details of how to create and include library projects are beyond the scope of this lesson, you can get up to speed quickly on their creation at the following links:

- Setting up a library project (Eclipse)
- Setting up a library project (Command line)

If you're converting an existing application to use multiple APK support, scour your codebase for every localized string file, list of values, theme colors, menu icons and layout that isn't going to change across APKs, and put it all in the library project. Code that isn't going to change much should also go in the library project. You'll likely find yourself extending these classes to add a method or two from APK to APK.

If, on the other hand, you're creating the application from scratch, try as much as possible to write code in the library project *first*, then only move it down to an individual APK if necessary. This is much easier to manage in the long run than adding it to one, then another, then another, then

months later trying to figure out whether this blob can be moved up to the library section without screwing anything up.

Create New APK Projects

There should be a separate Android project for each APK you're going to release. For easy organization, place the library project and all related APK projects under the same parent folder. Also remember that each APK needs to have the same package name, although they don't necessarily need to share the package name with the library. If you were to have 3 APKs following the scheme described earlier, your root directory might look like this:

```
alexlucas:~/code/multi-apks-root$ ls
foo-blue
foo-green
foo-lib
foo-red
```

Once the projects are created, add the library project as a reference to each APK project. If possible, define your starting Activity in the library project, and extend that Activity in your APK project. Having a starting activity defined in the library project gives you a chance to put all your application initialization in one place, so that each individual APK doesn't have to re-implement "universal" tasks like initializing Analytics, running licensing checks, and any other initialization procedures that don't change much from APK to APK.

Adjust the Manifests

When a user downloads an application which uses multiple APKs through Google Play, the correct APK to use is chosen using some simple rules:

- The manifest has to show that particular APK is eligible
- Of the eligible APKs, highest version number wins
- If *any* of the texture formats listed in your APK are supported by the device on market, that device is considered eligible

With regards to GL Textures, that last rule is important. It means that

you should, for instance, be *very* careful about using different GL formats in the same application. If you were to use PowerVR 99% of the time, but use ETC1 for, say, your splash screen... Then your manifest would necessarily indicate support for both formats. A device that *only* supported ETC1 would be deemed compatible, your app would download, and the user would see some thrilling crash messages. The common case is going to be that if you're using multiple APKs specifically to target different devices based on GL texture support, it's going to be one texture format per APK.

This actually makes texture support a little bit different than the other two multiple APK dimensions, API level and screen size. Any given device only has one API level, and one screen size, and it's up to the APK to support a range of them. With textures, the APK will generally support one texture, and the device will support many. There will often be overlap in terms of one device supporting many APKs, but the solution is the same: Version codes.

By way of example, take a few devices, and see how many of the APKs defined earlier fit each device.

FooPhone	Nexus S	Evo
ETC1	ETC1	ETC1
	PowerVR	ATI TC

Assuming that PowerVR and ATI formats are both preferred over ETC1 when available, than according to the "highest version number wins" rule, if we set the versionCode attribute in each APK such that red ≥ green ≥ blue, then both Red and Green will always be chosen over Blue on devices which support them, and should a device ever come along which supports both Red and Green, red will be chosen.

In order to keep all your APKs on separate "tracks," it's important to have a good version code scheme. The recommended one can be found on the Version Codes area of our developer guide. Since the example set of

APKs is only dealing with one of 3 possible dimensions, it would be sufficient to separate each APK by 1000 and increment from there. This might look like:

Blue: 1001, 1002, 1003, 1004...
Green: 2001, 2002, 2003, 2004...
Red:3001, 3002, 3003, 3004...

Putting this all together, your Android Manifests would likely look something like the following:

Blue:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="1001" android:versionName="1.0"
package="com.example.foo">
    <supports-gl-texture
android:name="GL_OES_compressed_ETC1_RGB8_texture" />
    ...
```

Green:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="2001" android:versionName="1.0"
package="com.example.foo">
    <supports-gl-texture
android:name="GL_AMD_compressed_ATC_texture" />
    ...
```

Red:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="3001" android:versionName="1.0"
package="com.example.foo">
    <supports-gl-texture
android:name="GL_IMG_texture_compression_pvrtc" />
    ...
```

Go Over Pre-launch Checklist

Before uploading to Google Play, double-check the following items. Remember that these are specifically relevant to multiple APKs, and in no way represent a complete checklist for all applications being uploaded to Google Play.

All APKs must have the same package name
- All APKs must be signed with the same certificate
- Double check your manifest filters for conflicting information (an APK that only supports cupcake on XLARGE screens isn't going to be seen by anybody)
- Each APK's manifest must be unique across at least one of supported screen, OpenGL texture, or platform version
- Try to test each APK on at least one device. Barring that, you have one of the most customizable device emulators in the business sitting on your development machine. Go nuts!

It's also worth inspecting the compiled APK before pushing to market, to make sure there aren't any surprises that could hide your application on Google Play. This is actually quite simple using the "aapt" tool. Aapt (the Android Asset Packaging Tool) is part of the build process for creating and packaging your Android applications, and is also a very handy tool for inspecting them.

```
>aapt dump badging
package: name='com.example.hello' versionCode='1' versionName='1.0'
sdkVersion:'11'
uses-permission:'android.permission.SEND_SMS'
application-label:'Hello'
application-icon-120:'res/drawable-ldpi/icon.png'
application-icon-160:'res/drawable-mdpi/icon.png'
application-icon-240:'res/drawable-hdpi/icon.png'
application: label='Hello' icon='res/drawable-mdpi/icon.png'
launchable-activity: name='com.example.hello.HelloActivity'
label='Hello' icon=''
uses-feature:'android.hardware.telephony'
uses-feature:'android.hardware.touchscreen'
main
supports-screens: 'xlarge'
supports-any-density: 'true'
locales: '--_--'
densities: '120' '160' '240'
```

When you examine aapt output, be sure to check that you don't have conflicting values for supports-screens and compatible-screens, and that you don't have unintended "uses-feature" values that were added as a result of permissions you set in the manifest. In the example above, the APK will be invisible to most, if not all devices.

Why? By adding the required permission SEND_SMS, the feature requirement of android.hardware.telephony was implicitly added. Since

most (if not all) xlarge devices are tablets without telephony hardware in them, Google Play will filter out this APK in these cases, until future devices come along which are both large enough to report as xlarge screen size, and possess telephony hardware.

Fortunately this is easily fixed by adding the following to your manifest:

```
<uses-feature android:name="android.hardware.telephony"
android:required="false" />
```

The `android.hardware.touchscreen` requirement is also implicitly added. If you want your APK to be visible on TVs which are non-touchscreen devices you should add the following to your manifest:

```
<uses-feature android:name="android.hardware.touchscreen"
android:required="false" />
```

Once you've completed the pre-launch checklist, upload your APKs to Google Play. It may take a bit for the application to show up when browsing Google Play, but when it does, perform one last check. Download the application onto any test devices you may have to make sure that the APKs are targeting the intended devices. Congratulations, you're done!

Next Get started Next class

Creating Multiple APKs with 2+ Dimensions

Previous Next Get started

This lesson teaches you to

1. Confirm You Need Multiple APKs
2. Chart Your Requirements
3. Put All Common Code and Resources in a Library Project.
4. Create New APK Projects
5. Adjust the Manifests
6. Go Over Pre-launch Checklist

You should also read

- Multiple APK Support

When developing your Android application to take advantage of multiple APKs on Google Play, it's important to adopt some good practices from the get-go, and prevent unnecessary headaches further into the development process. This lesson shows you how to create multiple APKs of your app, each covering a different class of screen size. You will also gain some tools necessary to make maintaining a multiple APK codebase as painless as possible.

Confirm You Need Multiple APKs

When trying to create an application that works across the huge range of available Android devices, naturally you want your application look its best on each individual device. You want to take advantage of the space of large screens but still work on small ones, to use new Android API

features or visual textures available on cutting edge devices but not abandon older ones. It may seem at the outset as though multiple APK support is the best solution, but this often isn't the case. The Using Single APK Instead section of the multiple APK guide includes some useful information on how to accomplish all of this with a single APK, including use of our support library, and links to resources throughout the Android Developer guide.

If you can manage it, confining your application to a single APK has several advantages, including:

- Publishing and Testing are easier
- There's only one codebase to maintain
- Your application can adapt to device configuration changes
- App restore across devices just works
- You don't have to worry about market preference, behavior from "upgrades" from one APK to the next, or which APK goes with which class of devices

The rest of this lesson assumes that you've researched the topic, studiously absorbed the material in the resources linked, and determined that multiple APKs are the right path for your application.

Chart Your Requirements

Start off by creating a simple chart to quickly determine how many APKs you need, and what screen size(s) each APK covers. Fortunately, it's easy to chart out your requirements quickly, easily, and have an easy reference for later. Let's say you want to split your APKs across two dimensions, API and screen size. Create a table with a row and column for each possible pair of values, and color in some "blobs", each color representing one APK.

	3	4	5	6	7	8	9	10	11	12	+
small											

Above is an example with four APKs. Blue is for all small/normal screen devices, Green is for large screen devices, and Red is for xlarge screen devices, all with an API range of 3-10. Purple is a special case, as it's for all screen sizes, but only for API 11 and up. More importantly, just by glancing at this chart, you immediately know which APK covers any given API/screen-size combo. To boot, you also have swanky codenames for each one, since "Have we tested red on the ?" is a lot easier to ask your cubie than "Have we tested the 3-to-10 xlarge APK against the Xoom?" Print this chart out and hand it to every person working on your codebase. Life just got a lot easier.

Put All Common Code and Resources in a Library Project.

Whether you're modifying an existing Android application or starting one from scratch, this is the first thing that you should do to the codebase, and by the far the most important. Everything that goes into the library project only needs to be updated once (think language-localized strings, color themes, bugs fixed in shared code), which improves your development time and reduces the likelihood of mistakes that could have been easily avoided.

Note: While the implementation details of how to create and include library projects are beyond the scope of this lesson, you can get up to speed quickly on their creation at the following links:

- Setting up a library project (Eclipse)
- Setting up a library project (Command line)

If you're converting an existing application to use multiple APK support,

scour your codebase for every localized string file, list of values, theme colors, menu icons and layout that isn't going to change across APKs, and put it all in the library project. Code that isn't going to change much should also go in the library project. You'll likely find yourself extending these classes to add a method or two from APK to APK.

If, on the other hand, you're creating the application from scratch, try as much as possible to write code in the library project *first*, then only move it down to an individual APK if necessary. This is much easier to manage in the long run than adding it to one, then another, then another, then months later trying to figure out whether this blob can be moved up to the library section without screwing anything up.

Create New APK Projects

There should be a separate Android project for each APK you're going to release. For easy organization, place the library project and all related APK projects under the same parent folder. Also remember that each APK needs to have the same package name, although they don't necessarily need to share the package name with the library. If you were to have 3 APKs following the scheme described earlier, your root directory might look like this:

```
alexlucas:~/code/multi-apks-root$ ls
foo-blue
foo-green
foo-lib
foo-purple
foo-red
```

Once the projects are created, add the library project as a reference to each APK project. If possible, define your starting Activity in the library project, and extend that Activity in your APK project. Having a starting activity defined in the library project gives you a chance to put all your application initialization in one place, so that each individual APK doesn't have to re-implement "universal" tasks like initializing Analytics, running licensing checks, and any other initialization procedures that don't change much from APK to APK.

Adjust the Manifests

When a user downloads an application which uses multiple APKs through Google Play, the correct APK to use is chosen using two simple rules:

- The manifest has to show that particular APK is eligible
- Of the eligible APKs, highest version number wins.

By way of example, let's take the set of multiple APKs described earlier, and assume that each APK has been set to support all screen sizes larger than its "target" screen size. Let's look at the sample chart from earlier:

	3	4	5	6	7	8	9	10	11	12	+
small											
normal											
large											
xlarge											

Since it's okay for coverage to overlap, we can describe the area covered by each APK like so:

- Blue covers all screens, minSDK 3.
- Green covers Large screens and higher, minSDK 3.
- Red covers XLarge screens (generally tablets), minSDK of 9.
- Purple covers all screens, minSDK of 11.

Note that there's a *lot* of overlap in those rules. For instance, an XLarge device with API 11 can conceivably run any one of the 4 APKs specified. However, by using the "highest version number wins" rule, we can set an order of preference as follows:

Purple ≥ Red ≥ Green ≥ Blue

Why allow all the overlap? Let's pretend that the Purple APK has some requirement on it that the other two don't. The Filters on Google Play page of the Android Developer guide has a whole list of possible culprits. For the sake of example, let's assume that Purple requires a front-facing camera. In fact, the entire point of Purple is to use entertaining things with the front-facing camera! But, it turns out, not all API 11+ devices even HAVE front-facing cameras! The horror!

Fortunately, if a user is browsing Google Play from one such device, Google Play will look at the manifest, see that Purple lists the front-facing camera as a requirement, and quietly ignore it, having determined that Purple and that device are not a match made in digital heaven. It will then see that Red is not only compatible with xlarge devices, but also doesn't care whether or not there's a front-facing camera! The app can still be downloaded from Google Play by the user, because despite the whole front-camera mishap, there was still an APK that supported that particular API level.

In order to keep all your APKs on separate "tracks", it's important to have a good version code scheme. The recommended one can be found on the Version Codes area of our developer guide. It's worth reading the whole section, but the basic gist is for this set of APKs, we'd use two digits to represent the minSDK, two to represent the min/max screen size, and 3 to represent the build number. That way, when the device upgraded to a new version of Android, (say, from 10 to 11), any APKs that are now eligible and preferred over the currently installed one would be seen by the device as an "upgrade". The version number scheme, when applied to the example set of APKs, might look like:

Blue: 0304001, 0304002, 0304003...
Green: 0334001, 0334002, 0334003
Red: 0344001, 0344002, 0344003...
Purple: 1104001, 1104002, 1104003...

Putting this all together, your Android Manifests would likely look something like the following:

Blue:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="0304001" android:versionName="1.0"
package="com.example.foo">
    <uses-sdk android:minSdkVersion="3" />
    <supports-screens android:smallScreens="true"
        android:normalScreens="true"
        android:largeScreens="true"
        android:xlargeScreens="true" />
    ...
```

Green:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="0334001" android:versionName="1.0"
package="com.example.foo">
    <uses-sdk android:minSdkVersion="3" />
    <supports-screens android:smallScreens="false"
        android:normalScreens="false"
        android:largeScreens="true"
        android:xlargeScreens="true" />
    ...
```

Red:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="0344001" android:versionName="1.0"
package="com.example.foo">
    <uses-sdk android:minSdkVersion="3" />
    <supports-screens android:smallScreens="false"
        android:normalScreens="false"
        android:largeScreens="false"
        android:xlargeScreens="true" />
    ...
```

Purple:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="1104001" android:versionName="1.0"
package="com.example.foo">
    <uses-sdk android:minSdkVersion="11" />
    <supports-screens android:smallScreens="true"
        android:normalScreens="true"
        android:largeScreens="true"
        android:xlargeScreens="true" />
    ...
```

Note that technically, multiple APK's will work with either the supports-screens tag, or the compatible-screens tag. Supports-screens is generally preferred, and it's generally a really bad idea to use both- It makes things needlessly complicated, and increases the opportunity for errors. Also note that instead of taking advantage of the default values (small and normal are always true by default), the manifests explicitly set the

value for each screen size. This can save you headaches down the line - By way of example, a manifest with a target SDK of < 9 will have xlarge automatically set to false, since that size didn't exist yet. So be explicit!

Go Over Pre-launch Checklist

Before uploading to Google Play, double-check the following items. Remember that these are specifically relevant to multiple APKs, and in no way represent a complete checklist for all applications being uploaded to Google Play.

- All APKs must have the same package name.
- All APKs must be signed with the same certificate.
- If the APKs overlap in platform version, the one with the higher minSdkVersion must have a higher version code.
- Every screen size you want your APK to support, set to true in the manifest. Every screen size you want it to avoid, set to false.
- Double check your manifest filters for conflicting information (an APK that only supports cupcake on XLARGE screens isn't going to be seen by anybody)
- Each APK's manifest must be unique across at least one of supported screen, OpenGL texture, or platform version.
- Try to test each APK on at least one device. Barring that, you have one of the most customizable device emulators in the business sitting on your development machine. Go nuts!

It's also worth inspecting the compiled APK before pushing to market, to make sure there aren't any surprises that could hide your application on Google Play. This is actually quite simple using the "aapt" tool. Aapt (the Android Asset Packaging Tool) is part of the build process for creating and packaging your Android applications, and is also a very handy tool for inspecting them.

```
>aapt dump badging
package: name='com.example.hello' versionCode='1' versionName='1.0'
sdkVersion:'11'
uses-permission:'android.permission.SEND_SMS'
application-label:'Hello'
application-icon-120:'res/drawable-ldpi/icon.png'
application-icon-160:'res/drawable-mdpi/icon.png'
```

```
application-icon-240:'res/drawable-hdpi/icon.png'
application: label='Hello' icon='res/drawable-mdpi/icon.png'
launchable-activity: name='com.example.hello.HelloActivity'
label='Hello' icon=''
uses-feature:'android.hardware.telephony'
uses-feature:'android.hardware.touchscreen'
main
supports-screens: 'xlarge'
supports-any-density: 'true'
locales: '--_--'
densities: '120' '160' '240'
```

When you examine aapt output, be sure to check that you don't have conflicting values for supports-screens and compatible-screens, and that you don't have unintended "uses-feature" values that were added as a result of permissions you set in the manifest. In the example above, the APK will be invisible to most, if not all devices.

Why? By adding the required permission SEND_SMS, the feature requirement of android.hardware.telephony was implicitly added. Since most (if not all) xlarge devices are tablets without telephony hardware in them, Google Play will filter out this APK in these cases, until future devices come along which are both large enough to report as xlarge screen size, and possess telephony hardware.

Fortunately this is easily fixed by adding the following to your manifest:

```
<uses-feature android:name="android.hardware.telephony"
android:required="false" />
```

The `android.hardware.touchscreen` requirement is also implicitly added. If you want your APK to be visible on TVs which are non-touchscreen devices you should add the following to your manifest:

```
<uses-feature android:name="android.hardware.touchscreen"
android:required="false" />
```

Once you've completed the pre-launch checklist, upload your APKs to Google Play. It may take a bit for the application to show up when browsing Google Play, but when it does, perform one last check. Download the application onto any test devices you may have to make sure that the APKs are targeting the intended devices. Congratulations, you're done!

Next Get started Next class

Creating a Custom Account Type

Previous Next Get started

This lesson teaches you to

1. Implement Your Custom Account Code
2. Be Smart About Security!
3. Extend AbstractAccountAuthenticator
4. Create an Authenticator Service
5. Distribute Your Service

You should also read

- SampleSyncAdapter app

So far we've talked about accessing Google APIs, which use accounts and users defined by Google. If you have your own online service, though, it won't have Google accounts or users, so what do you do? It turns out to be relatively straightforward to install new account types on a user's device. This lesson explains how to create a custom account type that works the same way as the built-in accounts do.

Implement Your Custom Account Code

The first thing you'll need is a way to get credentials from the user. This may be as simple as a dialog box that asks for a name and a password. Or it may be a more exotic procedure like a one-time password or a biometric scan. Either way, it's your responsibility to implement the code that:

1. Collects credentials from the user
2. Authenticates the credentials with the server

3. Stores the credentials on the device

Typically all three of these requirements can be handled by one activity. We'll call this the authenticator activity.

Because they need to interact with the `AccountManager` system, authenticator activities have certain requirements that normal activities don't. To make it easy to get things right, the Android framework supplies a base class, `AccountAuthenticatorActivity`, which you can extend to create your own custom authenticator.

How you address the first two requirements of an authenticator activity, credential collection and authentication, is completely up to you. (If there were only one way to do it, there'd be no need for "custom" account types, after all.) The third requirement has a canonical, and rather simple, implementation:

```
final Account account = new Account(mUsername, your_account_type);
mAccountManager.addAccountExplicitly(account, mPassword, null);
```

Be Smart About Security!

It's important to understand that `AccountManager` is not an encryption service or a keychain. It stores account credentials just as you pass them, in **plain text**. On most devices, this isn't a particular concern, because it stores them in a database that is only accessible to root. But on a rooted device, the credentials would be readable by anyone with `adb` access to the device.

With this in mind, you shouldn't pass the user's actual password to `AccountManager.addAccountExplicitly()`. Instead, you should store a cryptographically secure token that would be of limited use to an attacker. If your user credentials are protecting something valuable, you should carefully consider doing something similar.

Remember: When it comes to security code, follow the "Mythbusters" rule: don't try this at home! Consult a security professional before implementing any custom account code.

Now that the security disclaimers are out of the way, it's time to get back to work. You've already implemented the meat of your custom account code; what's left is plumbing.

Extend AbstractAccountAuthenticator

In order for the `AccountManager` to work with your custom account code, you need a class that implements the interfaces that `AccountManager` expects. This class is the *authenticator class*.

The easiest way to create an authenticator class is to extend `AbstractAccountAuthenticator` and implement its abstract methods. If you've worked through the previous lessons, the abstract methods of `AbstractAccountAuthenticator` should look familiar: they're the opposite side of the methods you called in the previous lesson to get account information and authorization tokens.

Implementing an authenticator class properly requires a number of separate pieces of code. First, `AbstractAccountAuthenticator` has seven abstract methods that you must override. Second, you need to add an intent filter for `"android.accounts.AccountAuthenticator"` to your application manifest (shown in the next section). Finally, you must supply two XML resources that define, among other things, the name of your custom account type and the icon that the system will display next to accounts of this type.

You can find a step-by-step guide to implementing a successful authenticator class and the XML files in the `AbstractAccountAuthenticator` documentation. There's also a sample implementation in the SampleSyncAdapter sample app.

As you read through the SampleSyncAdapter code, you'll notice that several of the methods return an intent in a bundle. This is the same intent that will be used to launch your custom authenticator activity. If your authenticator activity needs any special initialization parameters, you can attach them to the intent using `Intent.putExtra()`.

Create an Authenticator Service

Now that you have an authenticator class, you need a place for it to live. Account authenticators need to be available to multiple applications and work in the background, so naturally they're required to run inside a `Service`. We'll call this the authenticator service.

Your authenticator service can be very simple. All it needs to do is create an instance of your authenticator class in `onCreate()` and call `getIBinder()` in `onBind()`. The SampleSyncAdapter contains a good example of an authenticator service.

Don't forget to add a `<service>` tag to your manifest file and add an intent filter for the AccountAuthenticator intent and declare the account authenticator:

```
<service ...>
   <intent-filter>
      <action android:name="android.accounts.AccountAuthenticator" />
   </intent-filter>
   <meta-data android:name="android.accounts.AccountAuthenticator"
              android:resource="@xml/authenticator" />
</service>
```

Distribute Your Service

You're done! The system now recognizes your account type, right alongside all the big name account types like "Google" and "Corporate." You can use the **Accounts & Sync** Settings page to add an account, and apps that ask for accounts of your custom type will be able to enumerate and authenticate just as they would with any other account type.

Of course, all of this assumes that your account service is actually installed on the device. If only one app will ever access the service, then this isn't a big deal—just bundle the service in the app. But if you want your account service to be used by more than one app, things get trickier. You don't want to bundle the service with all of your apps and have multiple copies of it taking up space on your user's device.

One solution is to place the service in one small, special-purpose APK. When an app wishes to use your custom account type, it can check the device to see if your custom account service is available. If not, it can direct the user to Google Play to download the service. This may seem like a great deal of trouble at first, but compared with the alternative of re-entering credentials for every app that uses your custom account, it's refreshingly easy.

Next Get started Next class

Stories
WORLD NEWS
Lorem ipsum dolor sit amet, consectetur...
5 minutes ago
Lorem ipsum dolor sit amet, consectetur...
10 minutes ago
Lorem ipsum dolor sit amet, consectetur...
3 hours ago
Lorem ipsum dolor sit amet, consectetur...
10 hours ago
Story List
Stories
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
January 1, 2012
WASHINGTON, DC — Praesent vestibulum nisl sapien, eu convallis nisl. Nam id sodales elit. Vestibulum sollicitudin faucibus orci, nec consequat velit suscipit vitae. Praesent et condimentum massa. Etiam ipsum nulla, lacinia ut semper eu, aliquam eu lacus. Aenean aliquam magna eget libero aliquam quis dapibus sapien auctor.
Story View
Stories
Stories
Photos
Saved
Photos
Photos
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
January 1, 2012

Top-level Navigation
(Dropdown Tabs)
Photo List
Photo Detail
Photo Detail
(Fullscreen)
Saved
STORIES
PHOTOS
Lorem ipsum dolor sit amet, consectetur...
10 minutes ago
Lorem ipsum dolor sit amet, consectetur...
3 hours ago
Saved Stories
Saved
STORIES
PHOTOS
Saved Photos

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Photos January 1, 2012 Photos Stories Stories Photos Saved Lorem ipsum dolor sit amet, consectetur adipiscing elit. WASHINGTON, DC — Praesent vestibu-lum nisl sapien, eu convallis nisl. Nam id sodales elit. Vestibulum sollicitudin fauci-bus orci, nec consequat velit suscipit vitae. Praesent et condimentum massa. Etiam ipsum nulla, lacinia ut semper eu, aliquam eu lacus. Aenean aliquam magna eget libero aliquam quis dapibus sapien auctor. Stories January 1, 2012 Stories WORLD NEWS Lorem ipsum dolor sit amet, consectetur... 5 minutes ago Lorem ipsum dolor sit amet, consectetur... 10 minutes ago Lorem ipsum dolor sit amet, consectetur... 3 hours ago Lorem ipsum dolor sit amet, consectetur... 10 hours ago Saved STORIES PHOTOS Saved STORIES PHOTOS Lorem ipsum dolor sit amet, consectetur... 10 minutes ago Lorem ipsum dolor sit amet, consectetur... 3 hours ago Story List Photo List Photo Detail Saved Stories Saved Photos Photo Detail(Fullscreen) Top-level Navigation(Dropdown Tabs) Story View Example News ApplicationPhone Wireframes (Portrait) News

Alternatives
Stories
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Story List + View
Story List (Alternative)
Story View (Alternative)
Photos
Photo List
Photo View
Photo View (Fullscreen)
Saved
Stories
Photos

Saved

Saved Lorem ipsum dolor sit amet, consectetur adipiscing elit. 5 days ago Lorem ipsum dolor sit amet, consectetur adipiscing elit. 15 days ago Lorem ipsum dolor sit amet, consectetur adipiscing elit. 1 month ago Lorem ipsum dolor sit amet, consectetur adipiscing elit. 4 months ago Photos Stories Stories WORLD NEWS Lorem ipsum dolor sit amet, consectetur adipiscing elit. 5 minutes ago Lorem ipsum dolor sit amet, consectetur adipiscing elit. 10 minutes ago Lorem ipsum dolor sit amet, consectetur adipiscing elit. 3 hours ago Lorem ipsum dolor sit amet, consectetur adipiscing elit. 10 hours ago Lorem ipsum dolor sit amet, consectetur adipiscing elit. 11 hours ago Lorem ipsum dolor sit amet, consectetur adipiscing elit. 20 hours ago Lorem ipsum dolor sit amet, consectetur adipiscing elit. Lorem ipsum dolor sit amet, consectetur adipiscing elit. WASHINGTON, DC — Praesent vestibulum nisl sapien, eu convallis nisl. Nam id sodales elit. Vestibulum sollicitudin faucibus orci, nec consequat velit suscipit vitae. Praesent et con-dimentum massa. Etiam ipsum nulla, lacinia ut semper eu, aliquam eu lacus. Aenean ali-quam magna eget libero aliquam quis dapi-bus sapien auctor. Cras a ipsum in eros plac-erat tempor. Integer vitae mauris turpis, vitae gravida tellus. Praesent tincidunt orci et enim gravida non commodo turpis malesuada. Ut dapibus luctus dictum. Nam in ipsum erat. January 1, 2012 Stories Lorem ipsum dolor sit amet, consectetur adipiscing elit. WASHINGTON, DC — Praesent vestibulum nisl sapien, eu convallis nisl. Nam id sodales elit. Vestibulum sollicitudin faucibus orci, nec consequat velit suscipit vitae. Praesent et con-dimentum massa. Etiam ipsum nulla, lacinia ut semper eu, aliquam eu lacus. Aenean ali- quam magna eget libero aliquam quis dapi-bus sapien auctor. Cras a ipsum in eros plac-erat tempor. Integer vitae mauris turpis, vitae gravida tellus. Praesent tincidunt orci et enim gravida non commodo turpis malesuada. Ut dapibus luctus dictum. Nam in ipsum erat. January 1, 2012 Stories Lorem ipsum dolor sit amet, consectetur adipiscing elit. 5 minutes ago Lorem ipsum dolor sit amet, consectetur adipiscing elit. 10 minutes ago Lorem ipsum dolor sit amet, consectetur adipiscing elit. 3 hours ago Lorem ipsum dolor sit amet, consectetur adipiscing elit. 10 hours ago Lorem ipsum dolor sit amet, consectetur adipiscing elit. 11 hours ago Lorem ipsum dolor sit amet, consectetur adipiscing elit. 20 hours ago Lorem ipsum dolor sit amet, consectetur adipiscing elit. WORLD NEWS Lorem ipsum dolor sit amet, consectetur adipiscing elit. 5 minutes ago Lorem

ipsum dolor sit amet, consectetur adipiscing elit. 10 minutes ago Lorem ipsum dolor sit amet, consectetur adipiscing elit. 3 hours ago Lorem ipsum dolor sit amet, consectetur adipiscing elit. 10 hours ago Lorem ipsum dolor sit amet, consectetur adipiscing elit. 11 hours ago Lorem ipsum dolor sit amet, consectetur adipiscing elit. 20 hours ago Lorem ipsum dolor sit amet, consectetur adipiscing elit. SPORTS Lorem ipsum dolor sit amet, consectetur adipiscing elit. 5 minutes ago Lorem ipsum dolor sit amet, consectetur adipiscing elit. 10 minutes ago Lorem ipsum dolor sit amet, consectetur adipiscing elit. 3 hours ago Lorem ipsum dolor sit amet, consectetur adipiscing elit. 10 hours ago Lorem ipsum dolor sit amet, consectetur adipiscing elit. 11 hours ago Lorem ipsum dolor sit amet, consectetur adipiscing elit. 20 hours ago Lorem ipsum dolor sit amet, consectetur adipiscing elit. BUSINESS Photos Lorem ipsum dolor sit amet, consectetur adipiscing elit. January 1, 2012 Photos Example News ApplicationTablet Wireframes (Landscape) Story List + View Story List(Alternative) Story View(Alternative) Alternatives Photo List Saved Photo View Photo View(Fullscreen)