

Android API Levels

As you develop your application on Android, it's useful to understand the platform's general approach to API change management. It's also important to understand the API Level identifier and the role it plays in ensuring your application's compatibility with devices on which it may be installed.

The sections below provide information about API Level and how it affects your applications.

For information about how to use the "Filter by API Level" control available in the API reference documentation, see [Filtering the documentation](#) at the end of this document.

In this document

[What is API Level?](#)

[Uses of API Level in Android](#)

[Development Considerations](#)

[Application forward compatibility](#)

[Application backward compatibility](#)

[Selecting a platform version and API Level](#)

[Declaring a minimum API Level](#)

[Testing against higher API Levels](#)

[Using a Provisional API Level](#)

[Filtering the Documentation](#)

See also

[<uses-sdk>](#) manifest element

What is API Level?

API Level is an integer value that uniquely identifies the framework API revision offered by a version of the Android platform.

The Android platform provides a framework API that applications can use to interact with the underlying Android system. The framework API consists of:

- A core set of packages and classes
- A set of XML elements and attributes for declaring a manifest file
- A set of XML elements and attributes for declaring and accessing resources
- A set of Intents
- A set of permissions that applications can request, as well as permission enforcements included in the system

Each successive version of the Android platform can include updates to the Android application framework API that it delivers.

Updates to the framework API are designed so that the new API remains compatible with earlier versions of the API. That is, most changes in the API are additive and introduce new or replacement functionality. As parts of the API are upgraded, the older replaced parts are deprecated but are not removed, so that existing applications can still use them. In a very small number of cases, parts of the API may be modified or removed, although typically such changes are only needed to ensure API robustness and application or system security. All other API parts from earlier revisions are carried forward without modification.

The framework API that an Android platform delivers is specified using an integer identifier called "API Level". Each Android platform version supports exactly one API Level, although support is implicit for all earlier API Levels (down to API Level 1). The initial release of the Android platform provided API Level 1 and subsequent releases have incremented the API Level.

The following table specifies the API Level supported by each version of the Android platform.

Platform Version	API Level
------------------	-----------

Android 2.2	8
Android 2.1	7
Android 2.0.1	6
Android 2.0	5
Android 1.6	4
Android 1.5	3
Android 1.1	2
Android 1.0	1

Uses of API Level in Android

The API Level identifier serves a key role in ensuring the best possible experience for users and application developers:

- It lets the Android platform describe the maximum framework API revision that it supports
- It lets applications describe the framework API revision that they require
- It lets the system negotiate the installation of applications on the user's device, such that version-incompatible applications are not installed.

Each Android platform version stores its API Level identifier internally, in the Android system itself.

Applications can use a manifest element provided by the framework API — `<uses-sdk>` — to describe the minimum and maximum API Levels under which they are able to run, as well as the preferred API Level that they are designed to support. The element offers three key attributes:

- `android:minSdkVersion` — Specifies the minimum API Level on which the application is able to run. The default value is "1".
- `android:targetSdkVersion` — Specifies the API Level on which the application is designed to run. In some cases, this allows the application to use manifest elements or behaviors defined in the target API Level, rather than being restricted to using only those defined for the minimum API Level.
- `android:maxSdkVersion` — Specifies the maximum API Level on which the application is able to run. **Important:** Please read the [<uses-sdk>](#) documentation before using this attribute.

For example, to specify the minimum system API Level that an application requires in order to run, the application would include in its manifest a `<uses-sdk>` element with a `android:minSdkVersion` attribute. The value of `android:minSdkVersion` would be the integer corresponding to the API Level of the earliest version of the Android platform under which the application can run.

When the user attempts to install an application, or when revalidating an application after a system update, the Android system first checks the `<uses-sdk>` attributes in the application's manifest and compares the values against its own internal API Level. The system allows the installation to begin only if these conditions are met:

- If a `android:minSdkVersion` attribute is declared, its value must be less than or equal to the system's API Level integer. If not declared, the system assumes that the application requires API Level 1.
- If a `android:maxSdkVersion` attribute is declared, its value must be equal to or greater than the system's API Level integer. If not declared, the system assumes that the application has no maximum API Level. Please read the [<uses-sdk>](#) documentation for more information about how the system handles this attribute.

When declared in an application's manifest, a `<uses-sdk>` element might look like this:

```
<manifest>
  ...
  <uses-sdk android:minSdkVersion="5" />
  ...
</manifest>
```

The principal reason that an application would declare an API Level in `android:minSdkVersion` is to tell the Android system that it is using APIs that were *introduced* in the API Level specified. If the application were to be somehow installed on a platform with a lower API Level, then it would crash at run-time when it tried to access APIs that don't exist. The system prevents such an outcome by not allowing the application to be installed if the lowest API Level it requires is higher than that of the platform version on the target device.

For example, the [android.appwidget](#) package was introduced with API Level 3. If an application uses that API, it must declare a `android:minSdkVersion` attribute with a value of "3". The application will then be installable on platforms such as Android 1.5 (API Level 3) and Android 1.6 (API Level 4), but not on the Android 1.1 (API Level 2) and Android 1.0 platforms (API Level 1).

For more information about how to specify an application's API Level requirements, see the [<uses-sdk>](#) section of the manifest file documentation.

Development Considerations

The sections below provide information related to API level that you should consider when developing your application.

Application forward compatibility

Android applications are generally forward-compatible with new versions of the Android platform.

Because almost all changes to the framework API are additive, an Android application developed using any given version of the API (as specified by its API Level) is forward-compatible with later versions of the Android platform and higher API levels. The application should be able to run on all later versions of the Android platform, except in isolated cases where the application uses a part of the API that is later removed for some reason.

Forward compatibility is important because many Android-powered devices receive over-the-air (OTA) system updates. The user may install your application and use it successfully, then later receive an OTA update to a new version of the Android platform. Once the update is installed, your application will run in a new run-time version of the environment, but one that has the API and system capabilities that your application depends on.

In some cases, changes *below* the API, such those in the underlying system itself, may affect your application when it is run in the new environment. For that reason it's important for you, as the application developer, to understand how the application will look and behave in each system environment. To help you test your application on various versions of the Android platform, the Android SDK includes multiple platforms that you can download. Each platform includes a compatible system image that you can run in an AVD, to test your application.

Application backward compatibility

Android applications are not necessarily backward compatible with versions of the Android platform older than the version against which they were compiled.

Each new version of the Android platform can include new framework APIs, such as those that give applications access to new platform capabilities or replace existing API parts. The new APIs are accessible to applications when running on the new platform and, as mentioned above, also when running on later versions of the platform, as specified by API Level. Conversely, because earlier versions of the platform do not include the new APIs, applications that use the new APIs are

unable to run on those platforms.

Although it's unlikely that an Android-powered device would be downgraded to a previous version of the platform, it's important to realize that there are likely to be many devices in the field that run earlier versions of the platform. Even among devices that receive OTA updates, some might lag and might not receive an update for a significant amount of time.

Selecting a platform version and API Level

When you are developing your application, you will need to choose the platform version against which you will compile the application. In general, you should compile your application against the lowest possible version of the platform that your application can support.

You can determine the lowest possible platform version by compiling the application against successively lower build targets. After you determine the lowest version, you should create an AVD using the corresponding platform version (and API Level) and fully test your application. Make sure to declare a `android:minSdkVersion` attribute in the application's manifest and set its value to the API Level of the platform version.

Declaring a minimum API Level

If you build an application that uses APIs or system features introduced in the latest platform version, you should set the `android:minSdkVersion` attribute to the API Level of the latest platform version. This ensures that users will only be able to install your application if their devices are running a compatible version of the Android platform. In turn, this ensures that your application can function properly on their devices.

If your application uses APIs introduced in the latest platform version but does *not* declare a `android:minSdkVersion` attribute, then it will run properly on devices running the latest version of the platform, but *not* on devices running earlier versions of the platform. In the latter case, the application will crash at runtime when it tries to use APIs that don't exist on the earlier versions.

Testing against higher API Levels

After compiling your application, you should make sure to test it on the platform specified in the application's `android:minSdkVersion` attribute. To do so, create an AVD that uses the platform version required by your application. Additionally, to ensure forward-compatibility, you should run and test the application on all platforms that use a higher API Level than that used by your application.

The Android SDK includes multiple platform versions that you can use, including the latest version, and provides an updater tool that you can use to download other platform versions as necessary.

To access the updater, use the `android` command-line tool, located in the `<sdk>/tools` directory. You can launch the Updater by using the `android` command without specifying any options. You can also simply double-click the `android.bat` (Windows) or `android` (OS X/Linux) file. In ADT, you can also access the updater by selecting **Window > Android SDK and AVD Manager**.

To run your application against different platform versions in the emulator, create an AVD for each platform version that you want to test. For more information about AVDs, see [Android Virtual Devices](#). If you are using a physical device for testing, ensure that you know the API Level of the Android platform it runs. See the table at the top of this document for a list of platform versions and their API Levels.

Using a Provisional API Level

In some cases, an "Early Look" Android SDK platform may be available. To let you begin developing on the platform although the APIs may not be final, the platform's API Level integer will not be specified. You must instead use the platform's *provisional API Level* in your application manifest, in order to build applications against the platform. A provisional API Level is not an integer, but a string matching the codename of the unreleased platform version. The

provisional API Level will be specified in the release notes for the Early Look SDK release notes and is case-sensitive.

The use of a provisional API Level is designed to protect developers and device users from inadvertently publishing or installing applications based on the Early Look framework API, which may not run properly on actual devices running the final system image.

The provisional API Level will only be valid while using the Early Look SDK and can only be used to run applications in the emulator. An application using the provisional API Level can never be installed on an Android device. At the final release of the platform, you must replace any instances of the provisional API Level in your application manifest with the final platform's actual API Level integer.

Filtering the Reference Documentation by API Level

Reference documentation pages on the Android Developers site offer a "Filter by API Level" control in the top-right area of each page. You can use the control to show documentation only for parts of the API that are actually accessible to your application, based on the API Level that it specifies in the `android:minSdkVersion` attribute of its manifest file.

To use filtering, select the checkbox to enable filtering, just below the page search box. Then set the "Filter by API Level" control to the same API Level as specified by your application. Notice that APIs introduced in a later API Level are then grayed out and their content is masked, since they would not be accessible to your application.

Filtering by API Level in the documentation does not provide a view of what is new or introduced in each API Level — it simply provides a way to view the entire API associated with a given API Level, while excluding API elements introduced in later API Levels.

If you decide that you don't want to filter the API documentation, just disable the feature using the checkbox. By default, API Level filtering is disabled, so that you can view the full framework API, regardless of API Level.

Also note that the reference documentation for individual API elements specifies the API Level at which each element was introduced. The API Level for packages and classes is specified as "Since <api level>" at the top-right corner of the content area on each documentation page. The API Level for class members is specified in their detailed description headers, at the right margin.

Except as noted, this content is licensed under [Apache 2.0](#). For details and restrictions, see the [Content License](#).

Android 2.2 r1 - 23 Aug 2010 18:08

[Site Terms of Service](#) - [Privacy Policy](#) - [Brand Guidelines](#)

[↑ Go to top](#)

Market Filters

When a user searches or browses in Android Market, the results are filtered, and some applications might not be visible. For example, if an application requires a trackball (as specified in the manifest file), then Android Market will not show the app on any device that does not have a trackball.

The manifest file and the device's hardware and features are only part of how applications are filtered — filtering also depends on the country and carrier, the presence or absence of a SIM card, and other factors.

Changes to the Android Market filters are independent of changes to the Android platform itself. This document will be updated periodically to reflect any changes that occur.

How Filters Work in Android Market

Android Market uses the filter restrictions described below to determine whether to show your application to a user who is browsing or searching for applications on a given device. When determining whether to display your app, Market checks the device's hardware and software capabilities, as well as its carrier, location, and other characteristics. It then compares those against the restrictions and dependencies expressed by the application itself, in its manifest, `.apk`, and publishing details. If the application is compatible with the device according to the filter rules, Market displays the application to the user. Otherwise, Market hides your application from search results and category browsing.

You can use the filters described below to control whether Market shows or hides your application to users. You can request any combination of the available filters for your app — for example, you could set a `minSdkVersion` requirement of "4" and set `smallScreens="false"` in the app, then when uploading the app to Market you could target European countries (carriers) only. Android Market's filters would prevent the application from being visible on any device that did not match all three of these requirements.

A filtered app is not visible within Market, even if a user specifically requests the app by clicking a deep link that points directly to the app's ID within Market. All filtering restrictions are associated with an application's version and can change between versions. For example:

- If you publish a new version of your app with stricter restrictions, the app will not be visible to users for whom it is filtered, even if those users were able to see the previous version.
- If a user has installed your application and you publish an upgrade that makes the app invisible to the user, the user will not see that an upgrade is available.

Market filters quickview

- Android Market applies filters that let you control whether your app is shown to a user who is browsing or searching for apps.
- Filtering is determined by elements in an app's manifest file, aspects of the device being used, and other factors.

In this document

[How Filters Work in Android Market](#)
[Filtering based on Manifest File Elements](#)
[Other Filters](#)

See also

[Compatibility](#)
[<supports-screens>](#)
[<uses-configuration>](#)
[<uses-feature>](#)
[<uses-library>](#)
[<uses-permission>](#)
[<uses-sdk>](#)



Interested in publishing your app on
Android Market?
[Go to Android Market »](#)

Filtering based on Manifest Elements

Most Market filters are triggered by elements within an application's manifest file, [AndroidManifest.xml](#), although not everything in the manifest file can trigger filtering. The table below lists the manifest elements that you can use to trigger Android Market filtering, and explains how the filtering works.

Table 1. Manifest elements that trigger filtering on Market.

Manifest Element	Filter Name	How It Works
<supports-screens>	Screen Size	<p>An application indicates the screen sizes that it is capable of supporting by setting attributes of the <code><supports-screens></code> element. When the application is published, Market uses those attributes to determine whether to show the application to users, based on the screen sizes of their devices.</p> <p>As a general rule, Market assumes that the platform on the device can adapt smaller layouts to larger screens, but cannot adapt larger layouts to smaller screens. Thus, if an application declares support for "normal" screen size only, Market makes the application available to both normal- and large-screen devices, but filters the application so that it is not available to small-screen devices.</p> <p>If an application does not declare attributes for <code><supports-screens></code>, Market uses the default values for those attributes, which vary by API Level. Specifically:</p> <ul style="list-style-type: none"> In API level 3, the <code><supports-screens></code> element itself is undefined and no attributes are available. In this case, Market assumes that the application is designed for normal-size screens and shows the application to devices that have normal or large screens. This behavior is especially significant for applications that set their <code>android:minSdkVersion</code> to 3 or lower, since Market will filter them from small-screen devices by default. Such applications can enable support for small-screen devices by adding a <code>android:targetSdkVersion="4"</code> attribute to the <code><uses-sdk></code> element in their manifest files. For more information, see Strategies for Legacy Applications. In API Level 4, the defaults for all of the attributes is <code>"true"</code>. If an application does not declare a <code><supports-screens></code> element, Market assumes that the application is designed for all screen sizes and does not filter it from any devices. If the application does not declare one of the attributes, Market uses the default value of <code>"true"</code> and does not filter the app for devices of corresponding screen size. <p>Example 1 The manifest declares <code><uses-sdk android:minSdkVersion="3"></code> and does not include a <code><supports-screens></code> element. Result: Android Market will not show the app to a user of a small-screen device, but will show it to users of normal and large-screen devices, unless other filters apply.</p>

		<p>Example 2 The manifest declares <code><uses-sdk android:minSdkVersion="3" android:targetSdkVersion="4"></code> and does not include a <code><supports-screens></code> element. Result: Android Market will show the app to users on all devices, unless other filters apply.</p> <p>Example 3 The manifest declares <code><uses-sdk android:minSdkVersion="4"></code> and does not include a <code><supports-screens></code> element. Result: Android Market will show the app to all users, unless other filters apply.</p> <p>For more information on how to declare support for screen sizes in your application, see <supports-screens> and Supporting Multiple Screens.</p>
<uses-configuration>	Device Configuration: keyboard, navigation, touch screen	<p>An application can request certain hardware features, and Android Market will show the app only on devices that have the required hardware.</p> <p>Example 1 The manifest includes <code><uses-configuration android:reqFiveWayNav="true" /></code>, and a user is searching for apps on a device that does not have a five-way navigational control. Result: Android Market will not show the app to the user.</p> <p>Example 2 The manifest does not include a <code><uses-configuration></code> element. Result: Android Market will show the app to all users, unless other filters apply.</p> <p>For more details, see <uses-configuration>.</p>
<uses-feature>	Device Features (<code>name</code>)	<p>An application can require certain device features to be present on the device. This functionality was introduced in Android 2.0 (API Level 5).</p> <p>Example 1 The manifest includes <code><uses-feature android:name="android.hardware.sensor.light" /></code>, and a user is searching for apps on a device that does not have a light sensor. Result: Android Market will not show the app to the user.</p> <p>Example 2 The manifest does not include a <code><uses-feature></code> element. Result: Android Market will show the app to all users, unless other filters apply.</p> <p>For more details, see <uses-feature>.</p> <p><i>A note about camera:</i> If an application requests the CAMERA permission using the <uses-permission> element, Market assumes that the application requires the camera and all camera features (such as autofocus). For</p>

		<p>applications that require the camera and are designed to run on Android 1.5 (API Level 3), declaring the CAMERA permission is an effective way of ensuring that Market filters your app properly, since <code>uses-feature</code> filtering is not available to applications compiled against the Android 1.5 platform. For more details about requiring or requesting a camera, see the required attribute of <code><uses-feature></code>.</p>
	<p>OpenGL-ES Version (<code>openGLESVersion</code>)</p>	<p>An application can require that the device support a specific OpenGL-ES version using the <code><uses-feature android:openGLESVersion="int"></code> attribute.</p> <p>Example 1 An app requests multiple OpenGL-ES versions by specifying <code>openGLESVersion</code> multiple times in the manifest. Result: Market assumes that the app requires the highest of the indicated versions.</p> <p>Example 2 An app requests OpenGL-ES version 1.1, and a user is searching for apps on a device that supports OpenGL-ES version 2.0. Result: Android Market will show the app to the user, unless other filters apply. If a device reports that it supports OpenGL-ES version X, Market assumes that it also supports any version earlier than X.</p> <p>Example 3 A user is searching for apps on a device that does not report an OpenGL-ES version (for example, a device running Android 1.5 or earlier). Result: Android Market assumes that the device supports only OpenGL-ES 1.0. Market will only show the user apps that do not specify <code>openGLESVersion</code>, or apps that do not specify an OpenGL-ES version higher than 1.0.</p> <p>Example 4 The manifest does not specify <code>openGLESVersion</code>. Result: Android Market will show the app to all users, unless other filters apply.</p> <p>For more details, see uses-feature.</p>
<p><uses-library></p>	<p>Software Libraries</p>	<p>An application can require specific shared libraries to be present on the device.</p> <p>Example 1 An app requires the <code>com.google.android.maps</code> library, and a user is searching for apps on a device that does not have the <code>com.google.android.maps</code> library. Result: Android Market will not show the app to the user.</p> <p>Example 2 The manifest does not include a <code><uses-library></code> element. Result: Android Market will show the app to all users, unless other filters apply.</p> <p>For more details, see uses-library.</p>

<uses-permission>		<i>(See the note in the description of <uses-feature>, above.)</i>
<uses-sdk>	Minimum Framework Version (minSdkVersion)	<p>An application can require a minimum API level.</p> <p>Example 1 The manifest includes <code><uses-sdk android:minSdkVersion="3"></code>, and the app uses APIs that were introduced in API Level 3. A user is searching for apps on a device that has API Level 2. Result: Android Market will not show the app to the user.</p> <p>Example 2 The manifest does not include <code>minSdkVersion</code>, and the app uses APIs that were introduced in API Level 3. A user is searching for apps on a device that has API Level 2. Result: Android Market assumes that <code>minSdkVersion</code> is "1" and that the app is compatible with all versions of Android. Market shows the app to the user and allows the user to download the app. The app crashes at runtime.</p> <p>Because you want to avoid this second scenario, we recommend that you always declare a <code>minSdkVersion</code>. For details, see android:minSdkVersion.</p>
	Maximum Framework Version (maxSdkVersion)	<p><i>Deprecated.</i> Android 2.1 and later do not check or enforce the <code>maxSdkVersion</code> attribute, and the SDK will not compile if <code>maxSdkVersion</code> is set in an app's manifest. For devices already compiled with <code>maxSdkVersion</code>, Market will respect it and use it for filtering.</p> <p>Declaring <code>maxSdkVersion</code> is <i>not</i> recommended. For details, see android:maxSdkVersion.</p>

Other Filters

Android Market uses other application characteristics to determine whether to show or hide an application for a particular user on a given device, as described in the table below.

Table 2. Application and publishing characteristics that affect filtering on Market.

Filter Name	How It Works
Publishing Status	<p>Only published applications will appear in searches and browsing within Android Market.</p> <p>Even if an app is unpublished, it can be installed if users can see it in their Downloads area among their purchased, installed, or recently uninstalled apps.</p> <p>If an application has been suspended, users will not be able to reinstall or update it, even if it appears in their Downloads.</p>
Priced Status	<p>Not all users can see paid apps. To show paid apps, a device must have a SIM card and be running Android 1.1 or later, and it must be in a country (as determined by SIM carrier) in which paid apps are available.</p>

Country / Carrier Targeting	<p>When you upload your app to the Android Market, you can select specific countries to target. The app will only be visible to the countries (carriers) that you select, as follows:</p> <ul style="list-style-type: none">• A device's country is determined based on the carrier, if a carrier is available. If no carrier can be determined, the Market application tries to determine the country based on IP.• Carrier is determined based on the device's SIM (for GSM devices), not the current roaming carrier.
Native Platform	<p>An application that includes native libraries that target a specific platform (ARM EABI v7, for example) will only be visible on devices that support that platform. For details about the NDK and using native libraries, see What is the Android NDK?</p>
Forward-Locked Applications	<p>To forward lock an application, set copy protection to "On" when you upload the application to Market. Market will not show copy-protected applications on developer devices or unreleased devices.</p>

[↑ Go to top](#)

Except as noted, this content is licensed under [Apache 2.0](#). For details and restrictions, see the [Content License](#).

Android 2.2 r1 - 23 Aug 2010 18:08

[Site Terms of Service](#) - [Privacy Policy](#) - [Brand Guidelines](#)

App Install Location

Beginning with API Level 8, you can allow your application to be installed on the external storage (for example, the device's SD card). This is an optional feature you can declare for your application with the [android:installLocation](#) manifest attribute. If you do *not* declare this attribute, your application will be installed on the internal storage only and it cannot be moved to the external storage.

To allow the system to install your application on the external storage, modify your manifest file to include the [android:installLocation](#) attribute in the `<manifest>` element, with a value of either "preferExternal" or "auto". For example:

```
<manifest
xmlns:android="http://schemas.android.com
/apk/res/android"

android:installLocation="preferExternal"
... >
```

If you declare "preferExternal", you request that your application be installed on the external storage, but the system does not guarantee that your application will be installed on the external storage. If the external storage is full, the system will install it on the internal storage. The user can also move your application between the two locations.

If you declare "auto", you indicate that your application may be installed on the external storage, but you don't have a preference of install location. The system will decide where to install your application based on several factors. The user can also move your application between the two locations.

When your application is installed on the external storage:

- There is no effect on the application performance so long as the external storage is mounted on the device.
- The `.apk` file is saved on the external storage, but all private user data, databases, optimized `.dex` files, and extracted native code are saved on the internal device memory.
- The unique container in which your application is stored is encrypted with a randomly generated key that can be decrypted only by the device that originally installed it. Thus, an application installed on an SD card works for only one device.
- The user can move your application to the internal storage through the system settings.

Warning: When the user enables USB mass storage to share files with a computer or unmounts the SD card via the system settings, the external storage is unmounted from the device and all applications running on the external storage are immediately killed.

Quickview

- You can allow your application to install on the device's external storage.
- Some types of applications should **not** allow installation on the external storage.
- Installing on the external storage is ideal for large applications that are not tightly integrated with the system (most commonly, games).

In this document

[Backward Compatibility](#)

[Applications That Should NOT Install on External Storage](#)

[Applications That Should Install on External Storage](#)

See also

[<manifest>](#)

Backward Compatibility

The ability for your application to install on the external storage is a feature available only on devices running API Level 8 (Android 2.2) or greater. Existing applications that were built prior to API Level 8 will always install on the internal storage and cannot be moved to the external storage (even on devices with API Level 8). However, if your application is designed to support an API Level *lower than* 8, you can choose to support this feature for devices with API Level 8 or greater and still be compatible with devices using an API Level lower than 8.

To allow installation on external storage and remain compatible with versions lower than API Level 8:

1. Include the `android:installLocation` attribute with a value of "auto" or "preferExternal" in the [<manifest>](#) element.
2. Leave your `android:minSdkVersion` attribute as is (something *less than* "8") and be certain that your application code uses only APIs compatible with that level.
3. In order to compile your application, change your build target to API Level 8. This is necessary because older Android libraries don't understand the `android:installLocation` attribute and will not compile your application when it's present.

When your application is installed on a device with an API Level lower than 8, the `android:installLocation` attribute is ignored and the application is installed on the internal storage.

Caution: Although XML markup such as this will be ignored by older platforms, you must be careful not to use programming APIs introduced in API Level 8 while your `minSdkVersion` is less than "8", unless you perform the work necessary to provide backward compatibility in your code. For information about building backward compatibility in your application code, see the [Backward Compatibility](#) article.

Applications That Should NOT Install on External Storage

When the user enables USB mass storage to share files with their computer (or otherwise unmounts or removes the external storage), any application installed on the external storage and currently running is killed. The system effectively becomes unaware of the application until mass storage is disabled and the external storage is remounted on the device. Besides killing the application and making it unavailable to the user, this can break some types of applications in a more serious way. In order for your application to consistently behave as expected, you **should not** allow your application to be installed on the external storage if it uses any of the following features, due to the cited consequences when the external storage is unmounted:

Services

Your running [Service](#) will be killed and will not be restarted when external storage is remounted. You can, however, register for the [ACTION_EXTERNAL_APPLICATIONS_AVAILABLE](#) broadcast Intent, which will notify your application when applications installed on external storage have become available to the system again. At which time, you can restart your Service.

Alarm Services

Your alarms registered with [AlarmManager](#) will be cancelled. You must manually re-register any alarms when external storage is remounted.

Input Method Engines

Your [IME](#) will be replaced by the default IME. When external storage is remounted, the user can open system settings to enable your IME again.

Live Wallpapers

Your running [Live Wallpaper](#) will be replaced by the default Live Wallpaper. When external storage is remounted, the user can select your Live Wallpaper again.

Live Folders

Your [Live Folder](#) will be removed from the home screen. When external storage is remounted, the user can add your Live Folder to the home screen again.

App Widgets

Your [App Widget](#) will be removed from the home screen. When external storage is remounted, your App Widget will *not* be available for the user to select until the system resets the home application (usually not until a system reboot).

Account Managers

Your accounts created with [AccountManager](#) will disappear until external storage is remounted.

Sync Adapters

Your [AbstractThreadedSyncAdapter](#) and all its sync functionality will not work until external storage is remounted.

Device Administrators

Your [DeviceAdminReceiver](#) and all its admin capabilities will be disabled, which can have unforeseeable consequences for the device functionality, which may persist after external storage is remounted.

If your application uses any of the features listed above, you **should not** allow your application to install on external storage. By default, the system *will not* allow your application to install on the external storage, so you don't need to worry about your existing applications. However, if you're certain that your application should never be installed on the external storage, then you should make this clear by declaring [android:installLocation](#) with a value of "`internalOnly`". Though this does not change the default behavior, it explicitly states that your application should only be installed on the internal storage and serves as a reminder to you and other developers that this decision has been made.

Applications That Should Install on External Storage

In simple terms, anything that does not use the features listed in the previous section are safe when installed on external storage. Large games are more commonly the types of applications that should allow installation on external storage, because games don't typically provide additional services when inactive. When external storage becomes unavailable and a game process is killed, there should be no visible effect when the storage becomes available again and the user restarts the game (assuming that the game properly saved its state during the normal [Activity lifecycle](#)).

If your application requires several megabytes for the APK file, you should carefully consider whether to enable the application to install on the external storage so that users can preserve space on their internal storage.

Except as noted, this content is licensed under [Apache 2.0](#). For details and restrictions, see the [Content License](#).

Android 2.2 r1 - 23 Aug 2010 18:08

[Site Terms of Service](#) - [Privacy Policy](#) - [Brand Guidelines](#)

[↑ Go to top](#)

Android Supported Media Formats

The [Core Media Formats](#) table below describes the media format support built into the Android platform. Note that any given mobile device may provide support for additional formats or file types not listed in the table.

As an application developer, you are free to make use of any media codec that is available on any Android-powered device, including those provided by the Android platform and those that are device-specific.

Core Media Formats

Type	Format	Encoder	Decoder	Details	File Type(s) Supported
Audio	AAC LC/LTP		X	Mono/Stereo content in any combination of standard bit rates up to 160 kbps and sampling rates from 8 to 48kHz	3GPP (.3gp) and MPEG-4 (.mp4, .m4a). No support for raw AAC (.aac)
	HE-AACv1 (AAC+)		X		
	HE-AACv2 (enhanced AAC+)		X		
	AMR-NB	X	X	4.75 to 12.2 kbps sampled @ 8kHz	3GPP (.3gp)
	AMR-WB		X	9 rates from 6.60 kbit/s to 23.85 kbit/s sampled @ 16kHz	3GPP (.3gp)
	MP3		X	Mono/Stereo 8-320Kbps constant (CBR) or variable bit-rate (VBR)	MP3 (.mp3)
	MIDI		X	MIDI Type 0 and 1. DLS Version 1 and 2. XMF and Mobile XMF. Support for ringtone formats RTTTTL/RTX, OTA, and iMelody	Type 0 and 1 (.mid, .xmf, .mxmf). Also RTTTTL/RTX (.rtttl, .rtx), OTA (.ota), and iMelody (.imy)
	Ogg Vorbis		X		Ogg (.ogg)
	PCM/WAVE		X	8- and 16-bit linear PCM (rates up to limit of hardware)	WAVE (.wav)
Image	JPEG	X	X	Base+progressive	JPEG (.jpg)
	GIF		X		GIF (.gif)
	PNG		X		PNG (.png)
	BMP		X		BMP (.bmp)

Video	H.263	X	X		3GPP (.3gp) and MPEG-4 (.mp4)
	H.264 AVC		X		3GPP (.3gp) and MPEG-4 (.mp4)
	MPEG-4 SP		X		3GPP (.3gp)

[↑ Go to top](#)

Except as noted, this content is licensed under [Apache 2.0](#). For details and restrictions, see the [Content License](#).

Android 2.2 r1 - 23 Aug 2010 18:08

[Site Terms of Service](#) - [Privacy Policy](#) - [Brand Guidelines](#)

Intents List: Invoking Google Applications on Android Devices

The table below lists the intents that your application can send, to invoke Google applications on Android devices in certain ways. For each action/uri pair, the table describes how the receiving Google application handles the intent.

For more information about intents, see the [Intents and Intent Filters](#).

Note that this list is not comprehensive.

Target Application	Intent URI	Intent Action	Result
Browser	http://web_address https://web_address	VIEW	Open a browser window to the URL specified.
	"" (empty string) http://web_address https://web_address	WEB_SEARCH	Opens the file at the location on the device in the browser.
Dialer	tel: <i>phone_number</i>	CALL	<p>Calls the entered phone number. Valid telephone numbers as defined in the IETF RFC 3966 are accepted. Valid examples include the following:</p> <ul style="list-style-type: none"> tel:2125551212 tel: (212) 555 1212 <p>The dialer is good at normalizing some kinds of schemes: for example telephone numbers, so the schema described isn't strictly required in the Uri (URI string) factory. However, if you have not tried a schema or are unsure whether it can be handled, use the Uri.fromParts (scheme, ssp, fragment) factory instead.</p> <p>Note: This requires your application to request the following permission in your manifest: <code><uses-permission id="android.permission.CALL_PHONE" /></code></p>
	tel: <i>phone_number</i> voicemail:	DIAL	Dials (but does not actually initiate the call) the number given (or the stored voicemail on the phone). Telephone number normalization described for CALL applies to DIAL as well.
Google Maps	geo:latitude,longitude geo:latitude,longitude?z=zoom geo:0,0?q=my+street+address geo:0,0?q=business+near+city	VIEW	Opens the Maps application to the given location or query. The Geo URI scheme (not fully supported) is currently under development .

			<p>The <i>z</i> field specifies the zoom level. A zoom level of 1 shows the whole Earth, centered at the given <i>lat,lng</i>. A zoom level of 2 shows a quarter of the Earth, and so on. The highest zoom level is 23. A larger zoom level will be clamped to 23.</p>										
Google Streetview	<pre>google.streetview:cbll=<i>lat,lng</i>& cbp=1,<i>yaw</i>,,<i>pitch</i>,<i>zoom</i>& mz=<i>mapZoom</i></pre>	VIEW	<p>Opens the Street View application to the given location. The URI scheme is based on the syntax used for Street View panorama information in Google Maps URLs. The <i>cbll</i> field is required. The <i>cbp</i> and <i>mz</i> fields are optional.</p> <table border="1"> <tr> <td><i>lat</i></td> <td>latitude</td> </tr> <tr> <td><i>lng</i></td> <td>longitude</td> </tr> <tr> <td><i>yaw</i></td> <td> Panorama center-of-view in degrees clockwise from North. Note: The two commas after the <i>yaw</i> parameter are required. They are present for backwards-compatibility reasons. </td> </tr> <tr> <td><i>pitch</i></td> <td> Panorama center-of-view in degrees from -90 (look straight up) to 90 (look straight down.) </td> </tr> <tr> <td><i>zoom</i></td> <td> Panorama zoom. 1.0 = normal zoom, 2.0 = zoomed in 2x, 3.0 = zoomed in 4x, and so on. A zoom of 1.0 is 90 degree horizontal FOV for a nominal landscape mode 4 x 3 aspect ratio display. Android phones in portrait mode will adjust the zoom so that the vertical FOV is approximately the same as the landscape vertical FOV. This means that the horizontal FOV of an Android phone in portrait mode is much narrower than in landscape mode. This is done to minimize the fisheye lens effect that would be present if a 90 degree horizontal FOV was used in portrait mode. </td> </tr> </table>	<i>lat</i>	latitude	<i>lng</i>	longitude	<i>yaw</i>	Panorama center-of-view in degrees clockwise from North. Note: The two commas after the <i>yaw</i> parameter are required. They are present for backwards-compatibility reasons.	<i>pitch</i>	Panorama center-of-view in degrees from -90 (look straight up) to 90 (look straight down.)	<i>zoom</i>	Panorama zoom. 1.0 = normal zoom, 2.0 = zoomed in 2x, 3.0 = zoomed in 4x, and so on. A zoom of 1.0 is 90 degree horizontal FOV for a nominal landscape mode 4 x 3 aspect ratio display. Android phones in portrait mode will adjust the zoom so that the vertical FOV is approximately the same as the landscape vertical FOV. This means that the horizontal FOV of an Android phone in portrait mode is much narrower than in landscape mode. This is done to minimize the fisheye lens effect that would be present if a 90 degree horizontal FOV was used in portrait mode.
<i>lat</i>	latitude												
<i>lng</i>	longitude												
<i>yaw</i>	Panorama center-of-view in degrees clockwise from North. Note: The two commas after the <i>yaw</i> parameter are required. They are present for backwards-compatibility reasons.												
<i>pitch</i>	Panorama center-of-view in degrees from -90 (look straight up) to 90 (look straight down.)												
<i>zoom</i>	Panorama zoom. 1.0 = normal zoom, 2.0 = zoomed in 2x, 3.0 = zoomed in 4x, and so on. A zoom of 1.0 is 90 degree horizontal FOV for a nominal landscape mode 4 x 3 aspect ratio display. Android phones in portrait mode will adjust the zoom so that the vertical FOV is approximately the same as the landscape vertical FOV. This means that the horizontal FOV of an Android phone in portrait mode is much narrower than in landscape mode. This is done to minimize the fisheye lens effect that would be present if a 90 degree horizontal FOV was used in portrait mode.												

mapZoom

The map zoom of the map location associated with this panorama. This value is passed on to the Maps activity when the Street View "Go to Maps" menu item is chosen. It corresponds to the `z` parameter in the `geo:` intent.

Except as noted, this content is licensed under [Apache 2.0](#). For details and restrictions, see the [Content License](#).

Android 2.2 r1 - 23 Aug 2010 18:08

[Site Terms of Service](#) - [Privacy Policy](#) - [Brand Guidelines](#)

[↑ Go to top](#)

Glossary

The list below defines some of the basic terminology of the Android platform.

.apk file

Android application package file. Each Android application is compiled and packaged in a single file that includes all of the application's code (.dex files), resources, assets, and manifest file. The application package file can have any name but *must* use the `.apk` extension. For example: `myExampleAppname.apk`. For convenience, an application package file is often referred to as an ".apk".

Related: [Application](#).

.dex file

Compiled Android application code file.

Android programs are compiled into .dex (Dalvik Executable) files, which are in turn zipped into a single .apk file on the device. .dex files can be created by automatically translating compiled applications written in the Java programming language.

Action

A description of something that an Intent sender wants done. An action is a string value assigned to an Intent. Action strings can be defined by Android or by a third-party developer. For example, `android.intent.action.VIEW` for a Web URL, or `com.example.rumblr.SHAKE_PHONE` for a custom application to vibrate the phone.

Related: [Intent](#).

Activity

A single screen in an application, with supporting Java code, derived from the [Activity](#) class. Most commonly, an activity is visibly represented by a full screen window that can receive and handle UI events and perform complex tasks, because of the Window it uses to render its window. Though an Activity is typically full screen, it can also be floating or transparent.

adb

Android Debug Bridge, a command-line debugging application included with the SDK. It provides tools to browse the device, copy tools on the device, and forward ports for debugging. If you are developing in Eclipse using the ADT Plugin, adb is integrated into your development environment. See [Android Debug Bridge](#) for more information.

Application

From a component perspective, an Android application consists of one or more activities, services, listeners, and intent receivers. From a source file perspective, an Android application consists of code, resources, assets, and a single manifest. During compilation, these files are packaged in a single file called an application package file (.apk).

Related: [.apk](#), [Activity](#)

Canvas

A drawing surface that handles compositing of the actual bits against a Bitmap or Surface object. It has methods for standard computer drawing of bitmaps, lines, circles, rectangles, text, and so on, and is bound to a Bitmap or Surface. Canvas is the simplest, easiest way to draw 2D objects on the screen. However, it does not support hardware acceleration, as OpenGL ES does. The base class is [Canvas](#).

Related: [Drawable](#), [OpenGL ES](#).

Content Provider

A data-abstraction layer that you can use to safely expose your application's data to other applications. A content provider is built on the [ContentProvider](#) class, which handles content query strings of a specific format to return data in a specific format. See [Content Providers](#) topic for more information.

Related: [URI Usage in Android](#)

Dalvik

The Android platform's virtual machine. The Dalvik VM is an interpreter-only virtual machine that executes files in the Dalvik Executable (.dex) format, a format that is optimized for efficient storage and memory-mappable execution. The virtual machine is register-based, and it can run classes compiled by a Java language compiler that have been transformed into its native format using the included "dx" tool. The VM runs on top of Posix-compliant operating systems, which it relies on for underlying functionality (such as threading and low level memory management). The Dalvik core class library is intended to provide a familiar development base for those used to programming with Java Standard Edition, but it is geared specifically to the needs of a small mobile device.

DDMS

Dalvik Debug Monitor Service, a GUI debugging application included with the SDK. It provides screen capture, log dump, and process examination capabilities. If you are developing in Eclipse using the ADT Plugin, DDMS is integrated into your development environment. See [Dalvik Debug Monitor Server](#) to learn more about the program.

Dialog

A floating window that that acts as a lightweight form. A dialog can have button controls only and is intended to perform a simple action (such as button choice) and perhaps return a value. A dialog is not intended to persist in the history stack, contain complex layout, or perform complex actions. Android provides a default simple dialog for you with optional buttons, though you can define your own dialog layout. The base class for dialogs is [Dialog](#).

Related: [Activity](#).

Drawable

A compiled visual resource that can be used as a background, title, or other part of the screen. A drawable is typically loaded into another UI element, for example as a background image. A drawable is not able to receive events, but does assign various other properties such as "state" and scheduling, to enable subclasses such as animation objects or image libraries. Many drawable objects are loaded from drawable resource files — xml or bitmap files that describe the image. Drawable resources are compiled into subclasses of [android.graphics.drawable](#). For more information about drawables and other resources, see [Resources](#).

Related: [Resources](#), [Canvas](#)

Intent

An message object that you can use to launch or communicate with other applications/activities asynchronously. An Intent object is an instance of [Intent](#). It includes several criteria fields that you can supply, to determine what application/activity receives the Intent and what the receiver does when handling the Intent. Available criteria include include the desired action, a category, a data string, the MIME type of the data, a handling class, and others. An application sends an Intent to the Android system, rather than sending it directly to another application/activity. The application can send the Intent to a single target application or it can send it as a broadcast, which can in turn be handled by multiple applications sequentially. The Android system is responsible for resolving the best-available receiver for each Intent, based on the criteria supplied in the Intent and the Intent Filters defined by other applications. For more information, see [Intents and Intent Filters](#).

Related: [Intent Filter](#), [Broadcast Receiver](#).

Intent Filter

A filter object that an application declares in its manifest file, to tell the system what types of Intents each of its components is willing to accept and with what criteria. Through an intent filter, an application can express interest in specific data types, Intent actions, URI formats, and so on. When resolving an Intent, the system evaluates all of the available intent filters in all applications and passes the Intent to the application/activity that best matches the Intent and criteria. For more information, see [Intents and Intent Filters](#).

Related: [Intent](#), [Broadcast Receiver](#).

Broadcast Receiver

An application class that listens for Intents that are broadcast, rather than being sent to a single target application/activity. The system delivers a broadcast Intent to all interested broadcast receivers, which handle the Intent sequentially.

Related: [Intent](#), [Intent Filter](#).

Layout Resource

An XML file that describes the layout of an Activity screen.

Related: [Resources](#)

Manifest File

An XML file that each application must define, to describe the application's package name, version, components (activities, intent filters, services), imported libraries, and describes the various activities, and so on. See [The AndroidManifest.xml File](#) for complete information.

Nine-patch / 9-patch / Ninepatch image

A resizable bitmap resource that can be used for backgrounds or other images on the device. See [Nine-Patch Stretchable Image](#) for more information.

Related: [Resources](#).

OpenGL ES

Android provides OpenGL ES libraries that you can use for fast, complex 3D images. It is harder to use than a Canvas object, but better for 3D objects. The [android.opengl](#) and [javax.microedition.khronos.opengles](#) packages expose OpenGL ES functionality.

Related: [Canvas](#), [Surface](#)

Resources

Nonprogrammatic application components that are external to the compiled application code, but which can be loaded from application code using a well-known reference format. Android supports a variety of resource types, but a typical application's resources would consist of UI strings, UI layout components, graphics or other media files, and so on. An application uses resources to efficiently support localization and varied device profiles and states. For example, an application would include a separate set of resources for each supported local or device type, and it could include layout resources that are specific to the current screen orientation (landscape or portrait). For more information about resources, see [Resources and Assets](#). The resources of an application are always stored in the `res/*` subfolders of the project.

Service

An object of class [Service](#) that runs in the background (without any UI presence) to perform various persistent actions, such as playing music or monitoring network activity.

Related: [Activity](#)

Surface

An object of type [Surface](#) representing a block of memory that gets composited to the screen. A Surface holds a Canvas object for drawing, and provides various helper methods to draw layers and resize the surface. You should not use this class directly; use [SurfaceView](#) instead.

Related: [Canvas](#)

SurfaceView

A View object that wraps a Surface for drawing, and exposes methods to specify its size and format dynamically. A SurfaceView provides a way to draw independently of the UI thread for resource-intensive operations (such as games or camera previews), but it uses extra memory as a result. SurfaceView supports both Canvas and OpenGL ES graphics. The base class is [SurfaceView](#).

Related: [Surface](#)

Theme

A set of properties (text size, background color, and so on) bundled together to define various default display settings. Android provides a few standard themes, listed in [R.style](#) (starting with "Theme_").

URIs in Android

Android uses URI strings as the basis for requesting data in a content provider (such as to retrieve a list of contacts) and for requesting actions in an Intent (such as opening a Web page in a browser). The URI scheme and format is specialized according to the type of use, and an application can handle specific URI schemes and strings in any way it wants. Some URI schemes are reserved by system components. For example, requests for data from a content provider must use the `content://`. In an Intent, a URI using an `http://` scheme will be handled by the browser.

View

An object that draws to a rectangular area on the screen and handles click, keystroke, and other interaction events. A View is a base class for most layout components of an Activity or Dialog screen (text boxes, windows, and so on). It receives calls from its parent object (see [viewgroup](#), below) to draw itself, and informs its parent object about where and how big it would like to be (which may or may not be respected by the parent). For more information, see [View](#).

Related: [Viewgroup](#), [Widget](#)

Viewgroup

A container object that groups a set of child Views. The viewgroup is responsible for deciding where child views are positioned and how large they can be, as well as for calling each to draw itself when appropriate. Some viewgroups are invisible and are for layout only, while others have an intrinsic UI (for instance, a scrolling list box). Viewgroups are all in the [widget](#) package, but extend [ViewGroup](#).

Related: [View](#)

Widget

One of a set of fully implemented View subclasses that render form elements and other UI components, such as a text box or popup menu. Because a widget is fully implemented, it handles measuring and drawing itself and responding to screen events. Widgets are all in the [android.widget](#) package.

Window

In an Android application, an object derived from the abstract class [Window](#) that specifies the elements of a generic window, such as the look and feel (title bar text, location and content of menus, and so on). Dialog and Activity use an implementation of this class to render a window. You do not need to implement this class or use windows in your application.

Except as noted, this content is licensed under [Apache 2.0](#). For details and restrictions, see the [Content License](#).

[↑ Go to top](#)

Android 2.2 r1 - 23 Aug 2010 18:08

[Site Terms of Service](#) - [Privacy Policy](#) - [Brand Guidelines](#)