Android Developers

Android 8.0 Features and APIs

Android 8.0 (API level 26) introduces a variety of new features and capabilities for users and developers. This document highlights what's new for developers.

Make sure to also check out Android 8.0 Behavior Changes (https://developer.android.com/about/versions/o/android-8.0-changes.html) to learn about areas where platform changes may affect your apps.

User Experience

Picture-in-Picture mode

Android 8.0 (API level 26) allows activities to launch in picture-in-picture (PIP) mode. PIP is a special type of multi-window mode mostly used for video playback. PIP mode is already available for Android TV; Android 8.0 makes the feature available on other Android devices.

Picture-in-picture in Android 8.0.

When an activity is in PIP mode, it is in the paused state, but should continue showing content. For this reason, you should make sure your app does not pause playback in its `onPause()` (https://developer.android.com/reference/android/app/Activity.html#onPause()) handler. Instead, you should pause video in `onStop()` (https://developer.android.com/reference/android/app/Activity.html#onStop()), and resume playback in `onStart()` (https://developer.android.com/reference/android/app/Activity.html#onStart()). For more information, see Multi-Window Lifecycle (https://developer.android.com/guide/topics/ui/multi-window.html#lifecycle).

To specify that your activity can use PIP mode, set `android:supportsPictureInPicture` to true in the manifest. (Beginning with Android 8.0, you do not need to set `android:resizeableActivity` to true if you are supporting PIP mode, either on Android TV or on other Android devices; you only need to set `android:resizeableActivity` if your activity supports other multi-window modes.)

Android 8.0 (API level 26) introduces a new object, `PictureInPictureParams` (https://developer.android.com/reference/android/app/PictureInPictureParams.html), which you pass to PIP methods to specify how an activity should behave when it is in PIP mode. This object specifies properties such as the activity's preferred aspect ratio.

The existing PIP methods described in Adding Picture-in-picture (https://developer.android.com/training/tv/playback/picture-in-picture.html) can now be used on all Android devices, not just on Android TV. In addition, Android 8.0 provides the following methods to support PIP mode:

- `Activity.enterPictureInPictureMode(PictureInPictureParams args)` (https://developer.android.com/reference/android/app/Activity.html#enterPictureInPictureMode(android.app.PictureInPictureParams)): Places the activity in picture-in-picture mode. The activity's aspect ratio and other configuration settings are specified by ***args***. If any fields in ***args*** are empty, the system uses the values set the last time you called `Activity.setPictureInPictureParams()` (https://developer.android.com/reference/android/app/Activity.html#setPictureInPictureParams(android.app.PictureInPictureParams)).

  The specified activity is placed in a corner of the screen; the rest of the screen is filled with the previous activity that was on screen. The activity entering PIP mode goes into the paused state, but remains started. If the user taps the PIP activity, the system shows a menu for the user to interact with; no touch events reach the activity while it is in the PIP state.

- `Activity.setPictureInPictureParams()` (https://developer.android.com/reference/android/app/Activity.html#setPictureInPictureParams(android.app.PictureInPictureParams)): Updates an activity's PIP configuration settings. If the activity is currently in PIP mode, the settings are updated; this is useful if activity's aspect ratio changes. If the activity is not in PIP mode, these configuration settings are used regardless of the `enterPictureInPictureMode()` method that you call.

Notifications

In Android 8.0 (API level 26), we've redesigned notifications to provide an easier and more consistent way to manage notification behavior and settings. These changes include:

- Notification channels: Android 8.0 introduces notification channels that allow you to create a user-customizable channel for each type of notification you want to display. The user interface refers to notification channels as *notification categories*. To learn how to implement notification channels, see Managing notification channels (https://developer.android.com/guide/topics/ui/notifiers/notifications.html#ManageChannels).

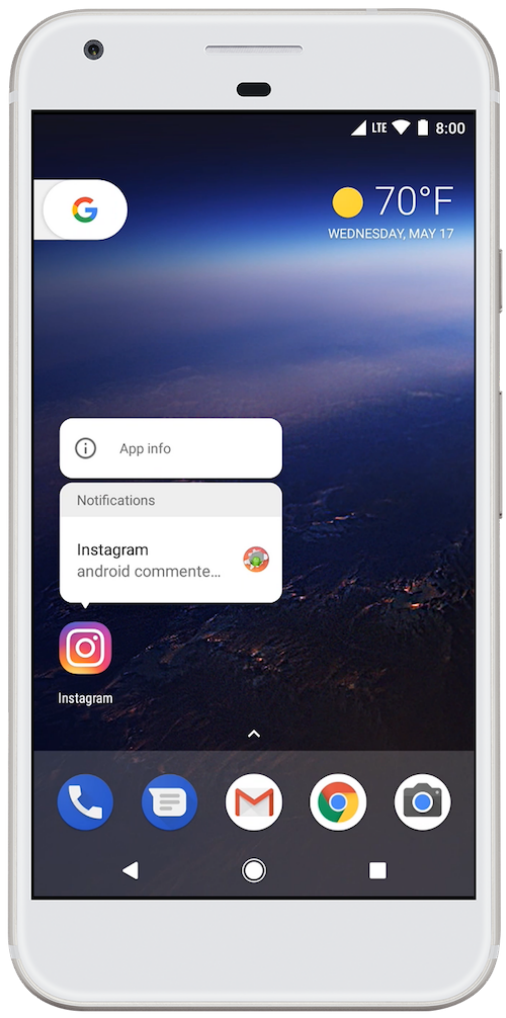
- Notification dots: Android 8.0 introduces support for displaying dots, or badges, on app launcher icons. Notification dots reflect the presence of notifications that the user has not yet dismissed or acted on. To learn how to work with notification dots, see Notification badges (https://developer.android.com/guide/topics/ui/notifiers/notifications.html#Badges).

- Snoozing: Users can snooze notifications, which causes them to disappear for a period of time before reappearing. Notifications reappear with the same level of importance they first appeared with. Apps can remove or update a snoozed notification, but updating a snoozed notification does not cause it to reappear.

- Notification timeouts: You can set a timeout when creating a notification using `setTimeoutAfter()` (https://developer.android.com/reference/android/app/Notification.Builder.html#setTimeoutAfter(long)). You can use this method to specify a duration after which a notification should be canceled. If required, you can cancel a notification before the specified timeout duration elapses.

- Notification settings: You can call `setSettingsText()` (https://developer.android.com/reference/android/app/Notification.Builder.html#setSettingsText(java.lang.CharSequence)) to set the text that appears when you create a link to your app's notification settings from a notification using the `Notification.INTENT_CATEGORY_NOTIFICATION_PREFERENCES` (https://developer.android.com/reference/android/app/Notification.html#INTENT_CATEGORY_NOTIFICATION_PREFERENCES) intent. The system may provide the following extras with the intent to filter the settings your app must display to users: `EXTRA_CHANNEL_ID`, `NOTIFICATION_TAG`, and `NOTIFICATION_ID`.

- Notification dismissal: Users can dismiss notifications themselves, and apps can remove them programmatically. You can determine when a notification is dismissed and why it's dismissed by implementing the `onNotificationRemoved()` (https://developer.android.com/reference/android/service/notification/NotificationListenerService.html#onNotificationRemoved(android.service.notification.StatusBarNotification)) method from the `NotificationListenerService` (https://developer.android.com/reference/android/service/notification/NotificationListenerService.html) class.



Users can long-press on app launcher icons to view notifications in Android 8.0.

- Background colors: You can set and enable a background color for a notification. You should only use this feature in notifications for ongoing tasks which are critical for a user to see at a glance. For example, you could set a background color for notifications related to driving directions, or a phone call in progress. You can also set the desired background color using `setColor()` (https://developer.android.com/reference/android/app/Notification.Builder.html#setColor(int)). Doing so allows you to use `setColorized()` (https://developer.android.com/reference/android/app/Notification.Builder.html#setColorized(boolean)) to enable the use of a background color for a notification.

- Messaging style: In Android 8.0, notifications that use the `MessagingStyle` (https://developer.android.com/reference/android/app/Notification.MessagingStyle.html) class display more content in their collapsed form. You should use the `MessagingStyle` (https://developer.android.com/reference/android/app/Notification.MessagingStyle.html) class for notifications that are messaging-related. You can also use the `addHistoricMessage()` (https://developer.android.com/reference/android/app/Notification.MessagingStyle.html#addHistoricMessage(android.app.Notification.MessagingStyle.Message)) method to provide context to a conversation by adding historic messages to messaging-related notifications.

Autofill framework

Account creation, login, and credit card transactions take time and are prone to errors. Users can easily get frustrated with apps that require these types of repetitive tasks.

Android 8.0 (API level 26) makes filling out forms, such as login and credit card forms, easier with the introduction of the Autofill Framework. Existing and new apps work with Autofill Framework after the user opts in to autofill.

You can take some steps to optimize how your app works with the framework. For more information, see Autofill Framework Overview (https://developer.android.com/guide/topics/text/autofill.html).

Downloadable fonts

Android 8.0 (API level 26) and Android Support Library 26 let you request fonts from a provider application instead of bundling fonts into the APK or letting the APK download fonts. This feature reduces your APK size, increases the app installation success rate, and allows multiple apps to share the same font.

For more information about downloading fonts, refer to Downloadable Fonts (https://developer.android.com/guide/topics/ui/look-and-feel/downloadable-fonts.html).

Fonts in XML

Android 8.0 (API level 26) introduces a new feature, Fonts in XML, which lets you use fonts as resources. This means, there is no need to bundle fonts as assets. Fonts are compiled in `R` file and are automatically available in the system as a resource. You can then access these fonts with the help of a new resource type, `font`.

The Support Library 26 provides full support to this feature on devices running API versions 14 and higher.

For more information, about using fonts as resources and retrieving system fonts, see Fonts in XML (https://developer.android.com/guide/topics/ui/look-and-feel/fonts-in-xml.html).

Autosizing TextView

Android 8.0 (API level 26) lets you set the size of your text expand or contract automatically based on the size of the TextView. This means, it is much easier to optimize the text size on different screens or with dynamic content. For more information, about autosizing TextView in Android 8.0, see Autosizing TextView (https://developer.android.com/guide/topics/ui/look-and-feel/autosizing-textview.html).

Adaptive icons

Android 8.0 (API level 26) introduces adaptive launcher icons. Adaptive icons support visual effects, and can display a variety of shapes across different device models. To learn how to create adaptive icons, see the Adaptive Icons (https://developer.android.com/guide/practices/ui_guidelines/icon_design_adaptive.html) guide.

Color management

Android developers of imaging apps can now take advantage of new devices that have a wide-gamut color capable display. To display wide gamut images, apps will need to enable a flag in their manifest (per activity) and load bitmaps with an embedded wide color profile (AdobeRGB, Pro Photo RGB, DCI-P3, etc.).

WebView APIs

Android 8.0 provides several APIs to help you manage the `WebView` `(https://developer.android.com/reference/android/webkit/WebView.html)` objects that display web content in your app. These APIs, which improve your app's stability and security, include the following:

- Version API
- Google SafeBrowsing API
- Termination Handle API
- Renderer Importance API

To learn more about how to use these APIs, see Managing WebViews (https://developer.android.com/guide/webapps/managing-webview.html).

The `WebView` `(https://developer.android.com/reference/android/webkit/WebView.html)` class now includes a Safe Browsing API to enhance the security of web browsing. For more information, see Google Safe Browsing API. (https://developer.android.com/guide/webapps/managing-webview.html#safe-browsing)

Pinning shortcuts and widgets

Android 8.0 (API level 26) introduces in-app pinning of shortcuts and widgets. In your app, you can create pinned shortcuts and widgets for supported launchers, subject to user permission.

For more information, see the Pinning Shortcuts and Widgets (https://developer.android.com/guide/topics/ui/shortcuts.html#pinning) feature guide.

Maximum screen aspect ratio

Android 8.0 (API level 26) brings changes to how to configure an app's maximum aspect ratio.

First, Android 8.0 introduces the maxAspectRatio (https://developer.android.com/reference/android/R.attr.html#maxAspectRatio) attribute, which you can use to set your app's maximum aspect ratio. In addition, in Android 8.0 and higher, an app's default maximum aspect ratio is the native aspect ratio of the device on which the app is running.

For more information about declaring maximum aspect ratio, see Supporting Multiple Screens (https://developer.android.com/guide/practices/screens_support.html).

Multi-display support

Beginning with Android 8.0 (API level 26), the platform offers enhanced support for multiple displays. If an activity supports multi-window mode and is running on a device with multiple displays, users can move the activity from one display to another. When an app launches an activity, the app can specify which display the activity should run on.

> **Note:** If an activity supports multi-window mode, Android 8.0 automatically enables multi-display support for that activity. You should test your app to make sure it works adequately in a multi-display environment.

Only one activity at a time can be in the resumed state, even if the app has multiple displays. The activity with focus is in the resumed state; all other visible activities are paused, but not stopped. For more information on the activity lifecycle when several activities are visible, see Multi-Window Lifecycle (https://developer.android.com/guide/topics/ui/multi-window.html#lifecycle).

When a user moves an activity from one display to another, the system resizes the activity and issues runtime changes as necessary. Your activity can handle the configuration change itself, or it can allow the system to destroy the process containing your activity and recreate it with the new dimensions. For more information, see Handling Configuration Changes (https://developer.android.com/guide/topics/resources/runtime-changes.html).

`ActivityOptions` (https://developer.android.com/reference/android/app/ActivityOptions.html) provides two new methods to support multiple displays:

`setLaunchDisplayId()` (https://developer.android.com/reference/android/app/ActivityOptions.html#setLaunchDisplayId(int))

Specifies which display the activity should be shown on when it is launched.

`getLaunchDisplayId()` (https://developer.android.com/reference/android/app/ActivityOptions.html#getLaunchDisplayId())

Returns the activity's current launch display.

The adb shell is extended to support multiple displays. The `shell start` command can now be used to launch an activity, and to specify the activity's target display:

```
adb shell start <activity_name> --display <display_id>
```

Unified layout margins and padding

Android 8.0 (API level 26) makes it easier for you to specify situations where opposite sides of a `View` (https://developer.android.com/reference/android/view/View.html) element use the same margin or padding. Specifically, you can now use the following attributes in your layout XML files:

- `layout_marginVertical` (https://developer.android.com/reference/android/R.attr.html#layout_marginVertical), which defines `layout_marginTop` (https://developer.android.com/reference/android/R.attr.html#layout_marginTop) and `layout_marginBottom` (https://developer.android.com/reference/android/R.attr.html#layout_marginBottom) at the same time.
- `layout_marginHorizontal` (https://developer.android.com/reference/android/R.attr.html#layout_marginHorizontal), which defines `layout_marginLeft` (https://developer.android.com/reference/android/R.attr.html#layout_marginLeft) and `layout_marginRight` (https://developer.android.com/reference/android/R.attr.html#layout_marginRight) at the same time.
- `paddingVertical` (https://developer.android.com/reference/android/R.attr.html#paddingVertical), which defines `paddingTop` (https://developer.android.com/reference/android/R.attr.html#paddingTop) and `paddingBottom` (https://developer.android.com/reference/android/R.attr.html#paddingBottom) at the same time.
- `paddingHorizontal` (https://developer.android.com/reference/android/R.attr.html#paddingHorizontal), which defines `paddingLeft` (https://developer.android.com/reference/android/R.attr.html#paddingLeft) and `paddingRight` (https://developer.android.com/reference/android/R.attr.html#paddingRight) at the same time.

> **Note:** If you customize your app's logic to support different languages and cultures (https://developer.android.com/training/basics/supporting-devices/languages.html), including text direction, keep in mind that these attributes don't affect the values of `layout_marginStart` (https://developer.android.com/reference/android/R.attr.html#layout_marginStart), `layout_marginEnd` (https://developer.android.com/reference/android/R.attr.html#layout_marginEnd), `paddingStart` (https://developer.android.com/reference/android/R.attr.html#paddingStart), or `paddingEnd` (https://developer.android.com/reference/android/R.attr.html#paddingEnd). You can set these values yourself, in addition to the new vertical and horizontal layout attributes, to create layout behavior that depends on the text direction.

Pointer capture

Some apps, such as games, remote desktop, and virtualization clients, greatly benefit from getting control over the mouse pointer. Pointer capture is a new feature in Android 8.0 (API level 26) that provides such control by delivering all mouse events to a focused view in your app.

Starting in Android 8.0, a `View` (https://developer.android.com/reference/android/view/View.html) in your app can request pointer capture and define a listener to process captured pointer events. The mouse pointer is hidden while in this mode. The view can release pointer capture when it doesn't need the mouse information anymore. The system can also release pointer capture when the view loses focus, for example, when the user opens another app.

For information on how to use this feature in your app, see Pointer capture (https://developer.android.com/training/gestures/movement.html#pointer-capture).

App categories

Android 8.0 (API level 26) allows each app to declare a category that it fits into, when relevant. These categories are used to cluster together apps of similar purpose or function when presenting them to users, such as in Data Usage, Battery Usage, or Storage Usage. You can define a category for your app by setting the `android:appCategory` attribute in your `<application>` manifest tag.

Android TV launcher

Android 8.0 (API level 26) includes a new content-centric, Android TV home screen experience (https://developer.android.com/training/tv/discovery/tvlauncher.html), which is available with the Android TV emulator and Nexus Player device image for Android 8.0. The new home screen organizes video content in rows corresponding to channels, which are each populated with programs by an app on the system. Apps can publish multiple channels, and users can configure which channels that they wish to see on the home screen. The Android TV home screen also includes a Watch Next row, which is populated with programs from apps, based on the viewing habits of the user. Apps can also provide video previews, which are automatically played when a user focuses on a program. The APIs for populating channels and programs are part of the TvProvider APIs, which are distributed as a Android Support Library module with Android 8.0.

AnimatorSet

Starting in Android 8.0 (API level 26), the `AnimatorSet` (https://developer.android.com/reference/android/animation/AnimatorSet.html) API now supports seeking and playing in reverse. Seeking lets you set the position of the animation set to a specific point in time. Playing in reverse is useful if your app includes animations for actions that can be undone. Instead of defining two separate animation sets, you can play the same one in reverse.

Input and navigation

Keyboard navigation clusters

If an activity in your app uses a complex view hierarchy, such as the one in Figure 2, consider organizing groups of UI elements into clusters for easier keyboard navigation among them. Users can press Meta+Tab, or Search+Tab on Chromebook devices, to navigate from one cluster to another. Good examples of clusters include: side panels, navigation bars, main content areas, and elements that could contain many child elements.

To make a `View` (https://developer.android.com/reference/android/view/View.html) or `ViewGroup` (https://developer.android.com/reference/android/view/ViewGroup.html) element a cluster, set the `android:keyboardNavigationCluster` (https://developer.android.com/reference/android/view/View.html#attr_android:keyboardNavigationCluster) attribute to `true` in the element's layout XML file, or pass `true` into `setKeyboardNavigationCluster()` (https://developer.android.com/reference/android/view/View.html#setKeyboardNavigationCluster(boolean)) in your app's UI logic.

> **Note:** Clusters cannot be nested, although non-nested clusters may appear at different levels of the hierarchy. If you attempt to nest clusters, the framework treats only the top-most `ViewGroup` (https://developer.android.com/reference/android/view/ViewGroup.html) element as a cluster.

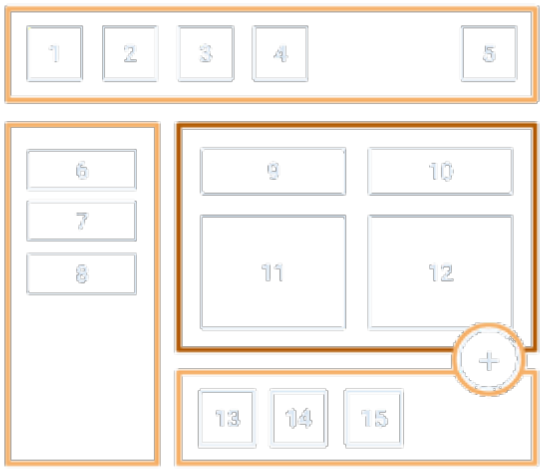


Figure 2. Activity containing 5 navigation clusters

On devices that have touchscreens, you can set a cluster-designated `ViewGroup` (https://developer.android.com/reference/android/view/ViewGroup.html) object's `android:touchscreenBlocksFocus` element to `true` to allow cluster-only navigation into and out of that cluster. If you apply this configuration to a cluster, users cannot use the Tab key or arrow keys to navigate into or out of the cluster; they must press the cluster navigation keyboard combination instead.

View default focus

In Android 8.0 (API level 26), you can assign the `View` (https://developer.android.com/reference/android/view/View.html) that should receive focus after a (re)created activity is resumed and the user presses a keyboard navigation key, such as the tab key. To apply this "focused by default" setting, set a `View` (https://developer.android.com/reference/android/view/View.html) element's `android:focusedByDefault` (https://developer.android.com/reference/android/view/View.html#attr_android:focusedByDefault) attribute to `true` in the layout XML file containing the UI element, or pass in `true` to `setFocusedByDefault()` (https://developer.android.com/reference/android/view/View.html#setFocusedByDefault(boolean)) in your app's UI logic.

System

New StrictMode detectors

Android 8.0 (API level 26) adds three new StrictMode detectors to help identify potential bugs in your app:

- `detectUnbufferedIo()` (https://developer.android.com/reference/android/os/StrictMode.ThreadPolicy.Builder.html#detectUnbufferedIo()) will detect when your app reads or writes data without buffering, which can drastically impact performance.
- `detectContentUriWithoutPermission()` (https://developer.android.com/reference/android/os/StrictMode.VmPolicy.Builder.html#detectContentUriWithoutPermission()) will detect when your app accidentally forgets to grant permissions to another app when starting an Activity outside your app.
- `detectUntaggedSockets()` (https://developer.android.com/reference/android/os/StrictMode.VmPolicy.Builder.html#detectUntaggedSockets()) will detect when your app performs network traffic without using `setThreadStatsTag(int)` (https://developer.android.com/reference/android/net/TrafficStats.html#setThreadStatsTag(int)) to tag your traffic for debugging purposes.

Cached data

Android 8.0 (API level 26) gives better guidance and behaviors around cached data. Each app is now given a disk space quota for cached data, as returned by `getCacheQuotaBytes(UUID)` (https://developer.android.com/reference/android/os/storage/StorageManager.html#getCacheQuotaBytes(java.util.UUID)).

When the system needs to free up disk space, it will start by deleting cached files from apps that are the most over their allocated quota. Thus, if you keep your cached data under your allocated quota, your cached files will be some of the last on the system to be cleared when necessary. When the system is deciding what cached files to delete inside your app, it will consider the oldest files first (as determined by modified time).

There are also two new behaviors that you can enable on a per-directory basis to control how the system frees up your cached data:

- `StorageManager.setCacheBehaviorAtomic()` can be used to indicate that a directory and all of its contents should be deleted as a single atomic unit.
- `setCacheBehaviorTombstone(File, boolean)` (https://developer.android.com/reference/android/os/storage

/StorageManager.html#setCacheBehaviorTombstone(java.io.File, boolean)) can be used to indicate that instead of deleting files inside a directory, they should be truncated to be 0 bytes in length, leaving the empty file intact.

Finally, when you need to allocate disk space for large files, consider using the new `allocateBytes(FileDescriptor, long)` (https://developer.android.com/reference/android/os/storage/StorageManager.html#allocateBytes(java.io.FileDescriptor, long)) API, which will automatically clear cached files belonging to other apps (as needed) to meet your request. When deciding if the device has enough disk space to hold your new data, call `getAllocatableBytes(UUID)` (https://developer.android.com/reference/android/os/storage/StorageManager.html#getAllocatableBytes(java.util.UUID)) instead of using `getUsableSpace()` (https://developer.android.com/reference/java/io/File.html#getUsableSpace()), since the former will consider any cached data that the system is willing to clear on your behalf.

Content provider paging

We've updated content providers to include support for loading a large dataset one page at a time. For example, a photo app with many thousands of images can query for a subset of the data to present in a page. Each page of results returned by a content provider is represented by a single Cursor object. Both a client and a provider must implement paging to make use of this feature.

For detailed information about the changes to content providers, see `ContentProvider` (https://developer.android.com/reference/android/content/ContentProvider.html) and `ContentProviderClient` (https://developer.android.com/reference/android/content/ContentProviderClient.html).

Content refresh requests

The `ContentProvider` (https://developer.android.com/reference/android/content/ContentProvider.html) and `ContentResolver` (https://developer.android.com/reference/android/content/ContentResolver.html) classes now each include a `refresh()` method, making it easier for clients to know whether the information they request is up-to-date.

You can add custom content refreshing logic by extending `ContentProvider` (https://developer.android.com/reference/android/content/ContentProvider.html). Make sure that you override the `refresh()` (https://developer.android.com/reference/android/content/ContentProvider.html#refresh(android.net.Uri, android.os.Bundle, android.os.CancellationSignal)) method to return `true`, indicating to your provider's clients that you've attempted to refresh the data yourself.

Your client app can explicitly request refreshed content by calling a different method, also called `refresh()` (https://developer.android.com/reference/android/content/ContentResolver.html#refresh(android.net.Uri, android.os.Bundle, android.os.CancellationSignal)). When calling this method, pass in the URI of the data to refresh.

> **Note:** Because you may be requesting data over a network, you should invoke `refresh()` (https://developer.android.com/reference/android/content/ContentResolver.html#refresh(android.net.Uri, android.os.Bundle, android.os.CancellationSignal)) from the client side only when there's a strong indication that the content is stale. The most common reason to perform this type of content refresh is in response to a swipe-to-refresh (https://developer.android.com/training/swipe/add-swipe-interface.html) gesture, explicitly requesting the current UI to display up-to-date content.

JobScheduler improvements

Android 8.0 (API level 26) introduces a number of improvements to `JobScheduler` (https://developer.android.com/reference/android/app/job/JobScheduler.html). These improvements make it easier for your app to comply with the new background execution limits (https://developer.android.com/about/versions/o/background.html), since you can generally use scheduled jobs to replace the now-restricted background services or implicit broadcast receivers.

Updates to `JobScheduler` (https://developer.android.com/reference/android/app/job/JobScheduler.html) include:

- You can now associate a work queue with a scheduled job. To add a work item to a job's queue, call `JobScheduler.enqueue()` (https://developer.android.com/reference/android/app/job/JobScheduler.html#enqueue(android.app.job.JobInfo,%20android.app.job.JobWorkItem)). When the job is running, it can take pending work off the queue and process it. This functionality handles many of the use cases that previously would have called for starting a background service, particulary services that implement `IntentService` (https://developer.android.com/reference/android/app/IntentService.html).
- Android Support Library 26.0.0 (https://developer.android.com/topic/libraries/support-library/revisions.html#26-0-0) introduces a

new `JobIntentService` (https://developer.android.com/reference/android/support/v4/app/JobIntentService.html) class, which provides the same functionality as `IntentService` (https://developer.android.com/reference/android/app/IntentService.html) but uses jobs instead of services when running on Android 8.0 (API level 26) or higher.

- You can now call `JobInfo.Builder.setClipData()` (https://developer.android.com/reference/android/app/job/JobInfo.Builder.html#setClipData(android.content.ClipData,%20int)) to associate a `ClipData` (https://developer.android.com/reference/android/content/ClipData.html) with a job. This option enables you to associate URI permission grants with a job, similarly to how these permissions can be propogated to `Context.startService()` (https://developer.android.com/reference/android/content/Context.html#startService(android.content.Intent)). You can also use URI permission grants with intents on work queues.
- Scheduled jobs now support several new constraints:

  `JobInfo.isRequireStorageNotLow()` (https://developer.android.com/reference/android/app/job/JobInfo.html#isRequireStorageNotLow())

  Job does not run if the device's available storage is low.

  `JobInfo.isRequireBatteryNotLow()` (https://developer.android.com/reference/android/app/job/JobInfo.html#isRequireBatteryNotLow())

  Job does not run if the battery level is at or below the criticial threshold; this is the level at which the device shows the **Low battery warning** system dialog.

  `NETWORK_TYPE_METERED` (https://developer.android.com/reference/android/app/job/JobInfo.html#NETWORK_TYPE_METERED)

  Job requires a metered network connection, like most cellular data plans.

Custom data store

Android 8.0 (API level 26) lets you provide a custom data store to your preferences, which can be useful if your app stores the preferences in a cloud or local database, or if the preferences are device-specific. For more information about implementing the data store, refer to Custom Data Store (https://developer.android.com/guide/topics/ui/settings.html).

Media enhancements

VolumeShaper

There is a new `VolumeShaper` (https://developer.android.com/reference/android/media/VolumeShaper.html) class. Use it to perform short automated volume transitions like fade-ins, fade-outs, and cross fades. See Controlling Amplitude with VolumeShaper (https://developer.android.com/guide/topics/media/volumeshaper.html) to learn more.

Audio focus enhancements

Audio apps share the audio output on a device by requesting and abandoning audio focus. An app handles changes in focus by starting or stopping playback, or ducking its volume. There is a new `AudioFocusRequest` (https://developer.android.com/reference/android/media/AudioFocusRequest.html) class. Using this class as the parameter of `requestAudioFocus()` (https://developer.android.com/reference/android/media/AudioManager.html#requestAudioFocus(android.media.AudioFocusRequest)), apps have new capabilites when handling changes in audio focus: automatic ducking (https://developer.android.com/guide/topics/media-apps/audio-focus.html#automatic-ducking) and delayed focus gain (https://developer.android.com/guide/topics/media-apps/audio-focus.html#delayed-focus-gain).

Media metrics

A new `getMetrics()` method returns a `PersistableBundle` (https://developer.android.com/reference/android/os/PersistableBundle.html) object containing configuration and performance information, expressed as a map of attributes and values. The `getMetrics()` method is defined for these media classes:

- `MediaPlayer.getMetrics()` (https://developer.android.com/reference/android/media/MediaPlayer.html#getMetrics())

- `MediaRecorder.getMetrics()` (https://developer.android.com/reference/android/media/MediaRecorder.html#getMetrics())
- `MediaCodec.getMetrics()` (https://developer.android.com/reference/android/media/MediaCodec.html#getMetrics())
- `MediaExtractor.getMetrics()` (https://developer.android.com/reference/android/media/MediaExtractor.html#getMetrics())

Metrics are collected separately for each instance and persist for the lifetime of the instance. If no metrics are available the method returns null. The actual metrics returned depend on the class.

MediaPlayer

Starting in Android 8.0 (API level 26) MediaPlayer can playback DRM-protected (https://developer.android.com/guide/topics/media/mediaplayer.html#drm) material and HLS sample-level encrypted media (https://developer.android.com/guide/topics/media/mediaplayer.html#encryption).

Android 8.0 introduces a new overloaded `seekTo()` (https://developer.android.com/reference/android/media/MediaPlayer.html#seekTo(long, int)) command that provides fine-grained control when seeking to a frame. It includes a second parameter that specifies a seek mode:

- `SEEK_PREVIOUS_SYNC` (https://developer.android.com/reference/android/media/MediaPlayer.html#SEEK_PREVIOUS_SYNC) moves the media position to a sync (or key) frame associated with a data source that is located right before or at the given time.
- `SEEK_NEXT_SYNC` (https://developer.android.com/reference/android/media/MediaPlayer.html#SEEK_NEXT_SYNC) moves the media position to a sync (or key) frame associated with a data source that is located right after or at the given time.
- `SEEK_CLOSEST_SYNC` (https://developer.android.com/reference/android/media/MediaPlayer.html#SEEK_CLOSEST_SYNC) moves the media position to a sync (or key) frame associated with a data source that is located closest to or at the given time.
- `SEEK_CLOSEST` (https://developer.android.com/reference/android/media/MediaPlayer.html#SEEK_CLOSEST) moves the media position to a frame (*not necessarily a sync or key frame*) associated with a data source that is located closest to or at the given time.

When seeking continuously, apps should use any of the `SEEK_` modes rather than `SEEK_CLOSEST` (https://developer.android.com/reference/android/media/MediaPlayer.html#SEEK_CLOSEST), which runs relatively slower but can be more precise.

MediaRecorder

- MediaRecorder now supports the MPEG2_TS format which is useful for streaming:

```
mMediaRecorder.setOutputFormat(MediaRecorder.OutputFormat.MPEG_2_TS);
```

  see `MediaRecorder.OutputFormat` (https://developer.android.com/reference/android/media/MediaRecorder.OutputFormat.html)

- The `MediaMuxer` (https://developer.android.com/reference/android/media/MediaMuxer.html) can now handle any number of audio and video streams. You are no longer limited to one audio track and/or one video track. Use `addTrack()` (https://developer.android.com/reference/android/media/MediaMuxer.html#addTrack(android.media.MediaFormat)) to mix as many tracks as you like.
- The `MediaMuxer` (https://developer.android.com/reference/android/media/MediaMuxer.html) can also add one or more metadata tracks containing user-defined per-frame information. The format of the metadata is defined by your application. The metadata track is only supported for MP4 containers.

Metadata can be useful for offline processing. For example, gyro signals from the sensor could be used to perform video stabilization.

When adding a metadata track, the track's mime format must start with the prefix "application/". Writing metadata is the same as writing video/audio data except that the data does not come from a `MediaCodec`. Instead, the app passes a `ByteBuffer` with an associated timestamp to the `writeSampleData()` (https://developer.android.com/reference/android/media/MediaMuxer.html#writeSampleData(int, java.nio.ByteBuffer,

android.media.MediaCodec.BufferInfo)) method. The timestamp must be in the same time base as the video and audio tracks.

The generated MP4 file uses the TextMetaDataSampleEntry defined in section 12.3.3.2 of the ISOBMFF to signal the metadata's mime format. When using MediaExtractor (https://developer.android.com/reference/android/media/MediaExtractor.html) to extract the file with metadata track, the mime format of the metadata will be extracted into MediaFormat (https://developer.android.com/reference/android/media/MediaFormat.html).

Improved media file access

The Storage Access Framework (SAF) (https://developer.android.com/guide/topics/providers/document-provider.html) allows apps to expose a custom DocumentsProvider (https://developer.android.com/reference/android/provider/DocumentsProvider.html), which can provide access to files in a data source to other apps. In fact, a documents provider can even provide access to files that reside on network storage or that use a protocol like Media Transfer Protocol (MTP) (https://en.wikipedia.org/wiki/Media_Transfer_Protocol).

However, accessing large media files from a remote data source introduces some challenges:

- Media players require seekable access to a file from a documents provider. In cases where a large media file resides on a remote data source, the documents provider must fetch all of the data in advance and create a snapshot file descriptor. The media player cannot play the file without the file descriptor, thus playback cannot begin until the documents provider finishes downloading the file.
- Media collection managers, such as photo apps, must traverse a series of access URIs to reach media that's stored on an external SD card via scoped folders. This access pattern makes mass operations on media—such as moving, copying, and deleting—quite slow.
- Media collection managers cannot determine a document's location given its URI. This makes it difficult for these types of apps to allow users to choose where to save a media file.

Android 8.0 addresses each of these challenges by improving the Storage Access Framework.

Custom document providers

Starting in Android 8.0, the Storage Access Framework allows custom documents providers (https://developer.android.com/guide/topics/providers/create-document-provider.html) to create seekable file descriptors for files residing in a remote data source. The SAF can open a file to get a native seekable file descriptor. The SAF then delivers discrete bytes requests to the documents provider. This feature allows a documents provider to return the exact range of bytes that a media player app has requested instead of caching the entire file in advance.

To use this feature, you need to call the new StorageManager.openProxyFileDescriptor() (https://developer.android.com/reference/android/os/storage/StorageManager.html#openProxyFileDescriptor(int, android.os.ProxyFileDescriptorCallback, android.os.Handler)) method. The openProxyFileDescriptor() (https://developer.android.com/reference/android/os/storage/StorageManager.html#openProxyFileDescriptor(int, android.os.ProxyFileDescriptorCallback, android.os.Handler)) method accepts a ProxyFileDescriptorCallback (https://developer.android.com/reference/android/os/ProxyFileDescriptorCallback.html) object as a callback. The SAF invokes the callback any time a client application performs file operations on the file descriptor returned from the documents provider.

Direct document access

As of Android 8.0 (API level 26), you can use the getDocumentUri() (https://developer.android.com/reference/android/provider/MediaStore.html#getDocumentUri(android.content.Context, android.net.Uri)) method to get a URI that references the same document as the given mediaUri. However, because the returned URI is backed by a DocumentsProvider (https://developer.android.com/reference/android/provider/DocumentsProvider.html), media collection managers can access the document directly, without having to traverse trees of scoped directories. As a result, the media managers can perform file operations on the document significantly more quickly.

> **Caution:** The getDocumentUri() (https://developer.android.com/reference/android/provider/MediaStore.html#getDocumentUri(android.content.Context, android.net.Uri)) method only locates media files; it doesn't grant apps permission to access those files. To learn more about how to obtain access permission to media files, see the reference documentation.

Paths to documents

When using the Storage Access Framework in Android 8.0 (API level 26), you can use the findDocumentPath() method, available in both the DocumentsContract (https://developer.android.com/reference/android/provider

/DocumentsContract.html#findDocumentPath(android.content.ContentResolver, android.net.Uri)) and DocumentsProvider (https://developer.android.com/reference/android/provider/DocumentsProvider.html#findDocumentPath(java.lang.String, java.lang.String)) classes, to determine the path from the root of a file system given a document's ID. The method returns this path in a DocumentsContract.Path (https://developer.android.com/reference/android/provider/DocumentsContract.Path.html) object. In cases where a file system has multiple defined paths to the same document, the method returns the path that is used most often to reach the document with the given ID.

This functionality is particularly useful in the following scenarios:

- Your app uses a "save as" dialog that displays the location of a particular document.
- Your app shows folders in a search results view and must load the child documents that are within a particular folder if the user selects that folder.

> **Note:** If your app has permission to access only some of the documents in the path, the return value of `findDocumentPath()` includes only the folders and documents that your app can access.

Monitoring audio playback

The AudioManager (https://developer.android.com/reference/android/media/AudioManager.html) system service maintains a list of active AudioPlaybackConfiguration (https://developer.android.com/reference/android/media/AudioPlaybackConfiguration.html) objects, each of which contains information about a particular audio playback session. Your app can retrieve the set of currently-active configurations by calling getActivePlaybackConfigurations() (https://developer.android.com/reference/android/media/AudioManager.html#getActivePlaybackConfigurations()).

As of Android 8.0 (API level 26), you can register a callback that notifies your app when one or more AudioPlaybackConfiguration (https://developer.android.com/reference/android/media/AudioPlaybackConfiguration.html) objects has changed. To do so, call registerAudioPlaybackCallback() (https://developer.android.com/reference/android/media/AudioManager.html#registerAudioPlaybackCallback(android.media.AudioManager.AudioPlaybackCallback, android.os.Handler)), passing in an instance of AudioManager.AudioPlaybackCallback (https://developer.android.com/reference/android/media/AudioManager.AudioPlaybackCallback.html). The AudioManager.AudioPlaybackCallback class contains the onPlaybackConfigChanged() (https://developer.android.com/reference/android/media/AudioManager.AudioPlaybackCallback.html#onPlaybackConfigChanged(java.util.List<android.media.AudioPlaybackConfiguration>)) method, which the system calls when the audio playback configuration changes.

Connectivity

Wi-Fi Aware

Android 8.0 (API level 26) adds support for Wi-Fi Aware, which is based on the Neighbor Awareness Networking (NAN) specification. On devices with the appropriate Wi-Fi Aware hardware, apps and nearby devices can discover and communicate over Wi-Fi without an Internet access point. We're working with our hardware partners to bring Wi-Fi Aware technology to devices as soon as possible. For information on how to integrate Wi-Fi Aware into your app, see Wi-Fi Aware (https://developer.android.com/guide/topics/connectivity/wifi-aware.html).

Bluetooth

Android 8.0 (API level 26) enriches the platform's Bluetooth support by adding the following features:

- Support for the AVRCP 1.4 standard, which enables song-library browsing.
- Support for the Bluetooth Low-Energy (BLE) 5.0 standard.
- Integration of the Sony LDAC codec into the Bluetooth stack.

Companion device pairing

Android 8.0 (API level 26) provides APIs that allow you to customize the pairing request dialog when trying to pair with companion devices over Bluetooth, BLE, and Wi-Fi. For more information, see Companion Device Pairing

(https://developer.android.com/guide/topics/connectivity/companion-device-pairing.html).

For more information about using Bluetooth on Android, see the Bluetooth (https://developer.android.com/guide/topics/connectivity/bluetooth.html) guide. For changes to Bluetooth that are specific to Android 8.0 (API level 26), see the Bluetooth (https://developer.android.com/about/versions/o/behavior-changes.html#bt) section of the Android 8.0 Behavior Changes (https://developer.android.com/about/versions/o/behavior-changes.html) page.

Sharing

Smart sharing

Android 8.0 (API level 26) learns about users' personalized sharing preferences and better understands for each type of content which are the right apps to share with. For example, if a user takes a photo of a receipt, Android 8.0 can suggest an expense-tracking app; if the user takes a selfie, a social media app can better handle the image. Android 8.0 automatically learns all these patterns according to users' personalized preferences.

Smart sharing works for types of content other than `image`, such as `audio`, `video`, `text`, `URL`, etc.

To enable Smart sharing, add an `ArrayList` (https://developer.android.com/reference/java/util/ArrayList.html) of up to three string annotations to the intent that shares the content. The annotations should describe the major components or topics in the content. The following code example shows how to add annotations to the intent:

```
ArrayList<String> annotations = new ArrayList<>();

annotations.add("topic1");
annotations.add("topic2");
annotations.add("topic3");

intent.putStringArrayListExtra(
    Intent.EXTRA_CONTENT_ANNOTATIONS,
    annotations
);
```

For detailed information about Smart sharing annotations, see `EXTRA_CONTENT_ANNOTATIONS` (https://developer.android.com/reference/android/content/Intent.html#EXTRA_CONTENT_ANNOTATIONS).

Text classifier

On compatible devices, apps can use a new Text Classifier to check whether a string matches a known classifier entity type and get suggested selection alternatives. Entities recognized by the system include addresses, URLs, telephone numbers, and email addresses. For more information, see `TextClassifier` (https://developer.android.com/reference/android/view/textclassifier/TextClassifier.html).

Accessibility

() () () () () () () ()

Android 8.0 (API level 26) supports several new accessibility features for developers who create their own accessibility services:

- A new volume category for adjusting accessibility volume (https://developer.android.com/guide/topics/ui/accessibility/services.html#volume).
- Fingerprint gestures (https://developer.android.com/guide/topics/ui/accessibility/services.html#fingerprint) as an input mechanism.
- Multilingual text to speech (https://developer.android.com/guide/topics/ui/accessibility/services.html#multilingual-tts) capabilities.
- A hardware-based accessibility shortcut (https://developer.android.com/guide/topics/ui/accessibility/services.html#shortcut) for quickly accessing a preferred accessibility service.
- Support for continued gestures (https://developer.android.com/guide/topics/ui/accessibility/services.html#continued-gestures), or programmatic sequences of strokes.

- An accessibility button (https://developer.android.com/guide/topics/ui/accessibility/services.html#button) for invoking one of several enabled accessibility features (available only on devices that use a software-rendered navigation area).
- Standardized one-sided range values (https://developer.android.com/guide/topics/ui/accessibility/services.html#one-sided-range).
- Several features for processing text (https://developer.android.com/guide/topics/ui/accessibility/services.html#process-text) more easily, including hint text, word-level dictation, and locations of on-screen text characters.

To learn more about how to make your app more accessible, see Accessibility (https://developer.android.com/guide/topics/ui/accessibility/index.html).

Security & Privacy

Permissions

Android 8.0 (API level 26) introduces several new permissions related to telephony:

- The `ANSWER_PHONE_CALLS` (https://developer.android.com/reference/android/Manifest.permission.html#ANSWER_PHONE_CALLS) permission allows your app to answer incoming phone calls programmatically. To handle an incoming phone call in your app, you can use the `acceptRingingCall()` (https://developer.android.com/reference/android/telecom/TelecomManager.html#acceptRingingCall()) method.
- The `READ_PHONE_NUMBERS` (https://developer.android.com/reference/android/Manifest.permission.html#READ_PHONE_NUMBERS) permission grants your app read access to the phone numbers stored in a device.

These permission are both classified as dangerous (https://developer.android.com/guide/topics/permissions/requesting.html#normal-dangerous) and are both part of the `PHONE` (https://developer.android.com/reference/android/Manifest.permission_group.html#PHONE) permission group.

New account access and discovery APIs

Android 8.0 (API level 26) introduces several improvements to how apps get access to user accounts. For the accounts that they manage, authenticators can use their own policy to decide whether to hide accounts from, or reveal accounts to, an app. The Android system tracks applications which can access a particular account.

In previous versions of Android, apps that wanted to track the list of user accounts had to get updates about all accounts, including accounts with unrelated types. Android 8.0 adds the `addOnAccountsUpdatedListener(android.accounts.OnAccountsUpdateListener, android.os.Handler, boolean, java.lang.String[])` (https://developer.android.com/reference/android/accounts/AccountManager.html#addOnAccountsUpdatedListener(android.accounts.OnAccountsUpdateListener, android.os.Handler, boolean, java.lang.String[])) method, which lets apps specify a list of account types for which account changes should be received.

API changes

AccountManager provides six new methods to help authenticators manage which apps can see an account:

- `setAccountVisibility(android.accounts.Account, java.lang.String, int)` (https://developer.android.com/reference/android/accounts/AccountManager.html#setAccountVisibility(android.accounts.Account, java.lang.String, int)): Sets the level of visibility for a specific user account and package combination.
- `getAccountVisibility(android.accounts.Account, java.lang.String)` (https://developer.android.com/reference/android/accounts/AccountManager.html#getAccountVisibility(android.accounts.Account, java.lang.String)): Gets the level of visibility for a specific user account and package combination.
- `getAccountsAndVisibilityForPackage(java.lang.String, java.lang.String)` (https://developer.android.com/reference/android/accounts/AccountManager.html#getAccountsAndVisibilityForPackage(java.lang.String, java.lang.String)): Allows authenticators to get the accounts and levels of visibility for a given package.
- `getPackagesAndVisibilityForAccount(android.accounts.Account)` (https://developer.android.com/reference/android/accounts/AccountManager.html#getPackagesAndVisibilityForAccount(android.accounts.Account)): Allows authenticators to get stored visibility values for a given account.

- `addAccountExplicitly(android.accounts.Account, java.lang.String, android.os.Bundle, java.util.Map<java.lang.String, java.lang.Integer>)` (https://developer.android.com/reference/android/accounts/AccountManager.html#addAccountExplicitly(android.accounts.Account, java.lang.String, android.os.Bundle, java.util.Map<java.lang.String, java.lang.Integer>)): Allows authenticators to initialize the visibility values of an account.
- `addOnAccountsUpdatedListener(android.accounts.OnAccountsUpdateListener, android.os.Handler, boolean, java.lang.String[])` (https://developer.android.com/reference/android/accounts/AccountManager.html#addOnAccountsUpdatedListener(android.accounts.OnAccountsUpdateListener, android.os.Handler, boolean, java.lang.String[])): Adds an `OnAccountsUpdateListener` (https://developer.android.com/reference/android/accounts/OnAccountsUpdateListener.html) listener to the `AccountManager` (https://developer.android.com/reference/android/accounts/AccountManager.html) object. The system calls this listener whenever the list of accounts on the device changes.

Android 8.0 (API level 26) introduces two special Package Name values to specify visibility levels for applications which were not set using the `setAccountVisibility(android.accounts.Account, java.lang.String, int)` (https://developer.android.com/reference/android/accounts/AccountManager.html#setAccountVisibility(android.accounts.Account, java.lang.String, int)) method. The `PACKAGE_NAME_KEY_LEGACY_VISIBLE` (https://developer.android.com/reference/android/accounts/AccountManager.html#PACKAGE_NAME_KEY_LEGACY_VISIBLE) visibility value is applied to apps that have the `GET_ACCOUNTS` (https://developer.android.com/reference/android/Manifest.permission.html#GET_ACCOUNTS) permission, and target versions of Android lower than Android 8.0, or whose signatures match the authenticator targeting any Android version. `PACKAGE_NAME_KEY_LEGACY_NOT_VISIBLE` (https://developer.android.com/reference/android/accounts/AccountManager.html#PACKAGE_NAME_KEY_LEGACY_NOT_VISIBLE) provides a default visibility value for apps which were not set previously and for which `PACKAGE_NAME_KEY_LEGACY_VISIBLE` (https://developer.android.com/reference/android/accounts/AccountManager.html#PACKAGE_NAME_KEY_LEGACY_VISIBLE) is not applicable.

For more information about the new account access and discovery APIs, see the reference for `AccountManager` (https://developer.android.com/reference/android/accounts/AccountManager.html) and `OnAccountsUpdateListener` (https://developer.android.com/reference/android/accounts/OnAccountsUpdateListener.html).

Testing

Instrumentation testing

Android 8.0 (API level 26) provides the following pieces of additional support for your app's instrumentation tests.

Run against non-default app processes

You can now specify that a particular instrumentation test should run against a process outside your app's default process. This configuration is useful if your app contains multiple activities that run in different processes.

To define non-default process instrumentation, navigate to your manifest file, then to the desired `<instrumentation>` (https://developer.android.com/guide/topics/manifest/instrumentation-element.html) element. Add the `android:targetProcess` attribute, and set its value to one of the following:

- The name of a particular process.
- A comma-separated list of process names.
- A wildcard (`"*"`), which allows the instrumentation to run against any launched process that executes code in the package specified in the `android:targetPackage` attribute.

While your instrumentation test is executing, you can check which process it's testing by calling `getProcessName()` (https://developer.android.com/reference/android/app/Instrumentation.html#getProcessName()).

Report results during a test

You can now report results while your instrumentation test is executing, rather than afterward, by calling `addResults()` (https://developer.android.com/reference/android/app/Instrumentation.html#addResults(android.os.Bundle)).

Mock intents for tests

To make it easier to create isolated, independent UI tests for your app's activities, Android 8.0 (API level 26) introduces the `onStartActivity()` (https://developer.android.com/reference/android/app/Instrumentation.ActivityMonitor.html#onStartActivity(android.content.Intent)) method. You override this method in a custom subclass of the `Instrumentation.ActivityMonitor` (https://developer.android.com/reference/android/app/Instrumentation.ActivityMonitor.html) class to handle a particular intent that your test class invokes.

When your test class invokes the intent, the method returns a stub `Instrumentation.ActivityResult` (https://developer.android.com/reference/android/app/Instrumentation.ActivityResult.html) object instead of executing the intent itself. By using this mock intent logic in your tests, you can focus on how your activity prepares and handles the intent that you pass to a different activity or to an entirely different app.

Runtime & Tools

Platform optimizations

Android 8.0 (API level 26) brings runtime and other optimizations to the platform that result in a number of performance improvements. These optimizations include concurrent-compaction garbage collection, more efficient use of memory, and code locality.

These optimizations result in faster boot times, as well as better performance in both the OS and apps.

Updated Java language support

Android 8.0 (API level 26) adds support for several additional OpenJDK Java APIs:

- `java.time` (https://developer.android.com/reference/java/time/package-summary.html) from OpenJDK 8.
- `java.nio.file` (https://developer.android.com/reference/java/nio/file/package-summary.html) and `java.lang.invoke` (https://developer.android.com/reference/java/lang/invoke/package-summary.html) from OpenJDK 7.

To learn more about the classes and methods within these newly-added packages, see the API reference documentation.

If you want to use Java 8 language features (https://developer.android.com/studio/preview/features/java8-support.html) in Android Studio, you should download the latest preview version (https://developer.android.com/studio/preview/index.html).

Updated ICU4J Android Framework APIs

Android 8.0 (API level 26) extends the ICU4J Android Framework APIs (https://developer.android.com/guide/topics/resources/icu4j-framework.html)—which is a subset of the ICU4J APIs—for app developers to use under the `android.icu` package. These APIs use localization data present on the device, so you can reduce your APK footprint by not compiling the ICU4J libraries in your APK.

Table 1. ICU, CLDR, and Unicode versions used in Android.

Android API level	ICU version	CLDR version	Unicode version
Android 7.0 (API level 24), Android 7.1 (API level 25)	56	28	8.0
Android 8.0 (API level 26)	58.2	30.0.3	9.0

For more information about internationalization on Android, including ICU4J support, see Internationalization on Android (https://developer.android.com/guide/topics/resources/internationalization.html).

Android enterprise

New enterprise features and APIs have been introduced for devices running Android 8.0 (API level 26). Highlights include the following:

- Work profiles on fully managed devices let enterprises separate work from personal data, while managing both.
- API delegation allows device owners and profile owners to assign app management to other applications.

- User experience improvements in the provisioning flow (including new customization options) reduce the setup time.
- New controls over Bluetooth, Wi-Fi, backup, and security let enterprises manage more of the device. Network activity logging help enterprises track down problems.

To learn more about these and other new Android enterprise APIs and features, see Android in the Enterprise (https://developer.android.com/work/versions/o.html).