ActivityManager | API reference | Android Developers

Archived: 2026-04-02 10:41:17 UTC

ActivityManager Stay organized with collections Save and categorize content based on your preferences.

`public class ActivityManager`
`extends Object`

This class gives information about, and interacts with, activities, services, and the containing process.

A number of the methods in this class are for debugging or informational purposes and they should not be used to affect any runtime behavior of your app. These methods are called out as such in the method level documentation.

Most application developers should not have the need to use this class, most of whose methods are for specialized use cases. However, a few methods are more broadly applicable. For instance, `isLowRamDevice()` enables your app to detect whether it is running on a low-memory device, and behave accordingly. `clearApplicationUserData()` is for apps with reset-data functionality.

In some special use cases, where an app interacts with its Task stack, the app may use the `ActivityManager.AppTask` and `ActivityManager.RecentTaskInfo` inner classes. However, in general, the methods in this class should be used for testing and debugging purposes only.

Summary

Public methods	
`int`	`addAppTask(Activity activity, Intent intent, ActivityManager.TaskDescription description, Bitmap thumbnail)` Add a new `AppTask` for the calling application.
`void`	`addApplicationStartInfoCompletionListener(Executor executor, Consumer<ApplicationStartInfo> listener)` Adds a callback that is notified when the `ApplicationStartInfo` record of this startup is complete.

`void`	`addStartInfoTimestamp(int key, long timestampNs)` Adds an optional developer supplied timestamp to the calling apps most recent `ApplicationStartInfo` .
`void`	`appNotResponding(String reason)` Method for the app to tell system that it's wedged and would like to trigger an ANR.
`boolean`	`clearApplicationUserData()` Permits an application to erase its own data from disk.
`void`	`clearWatchHeapLimit()` Clear a heap watch limit previously set by `setWatchHeapLimit(long)` .
`void`	`dumpPackageState(FileDescriptor fd, String packageName)` Perform a system dump of various state associated with the given application package name.
`Size`	`getAppTaskThumbnailSize()` Return the current design dimensions for `AppTask` thumbnails, for use with `addAppTask(Activity, Intent, TaskDescription, Bitmap)` .
`List<ActivityManager.AppTask>`	`getAppTasks()` Get the list of tasks associated with the calling application.
`ConfigurationInfo`	`getDeviceConfigurationInfo()` Get the device configuration attributes.
`List<ApplicationExitInfo>`	`getHistoricalProcessExitReasons(String packageName, int pid, int maxNum)` Return a list of `ApplicationExitInfo` records containing the reasons for the most recent app deaths.

`List<ApplicationStartInfo>`	`getHistoricalProcessStartReasons(int maxNum)` Return a list of `ApplicationStartInfo` records containing the information about the most recent app startups.
`int`	`getLargeMemoryClass()` Return the approximate per-application memory class of the current device when an application is running with a large heap.
`int`	`getLauncherLargeIconDensity()` Get the preferred density of icons for the launcher.
`int`	`getLauncherLargeIconSize()` Get the preferred launcher icon size.
`int`	`getLockTaskModeState()` Return the current state of task locking.
`int`	`getMemoryClass()` Return the approximate per-application memory class of the current device.
`void`	`getMemoryInfo(ActivityManager.MemoryInfo outInfo)` Return general information about the memory state of the system.
`static void`	`getMyMemoryState(ActivityManager.RunningAppProcessInfo outState)` Return global memory state information for the calling process.
`MemoryInfo[]`	`getProcessMemoryInfo(int[] pids)` Return information about the memory usage of one or more processes.
`List<ActivityManager.ProcessErrorStateInfo>`	`getProcessesInErrorState()` Returns a list of any processes that are currently in an error condition.
`List<ActivityManager.RecentTask`	`getRecentTasks(int maxNum, int flags)`

Info>	*This method was deprecated in API level 21. As of* `Build.VERSION_CODES.LOLLIPOP`, *this method is no longer available to third party applications: the introduction of document-centric recents means it can leak personal information to the caller. For backwards compatibility, it will still return a small subset of its data: at least the caller's own tasks (though see* `getAppTasks()` *for the correct supported way to retrieve that information), and possibly some other tasks such as home that are known to not be sensitive.*
`List<ActivityManager.RunningAppProcessInfo>`	`getRunningAppProcesses()` Returns a list of application processes that are running on the device.
`PendingIntent`	`getRunningServiceControlPanel(ComponentName service)` Returns a PendingIntent you can start to show a control panel for the given running service.
`List<ActivityManager.RunningServiceInfo>`	`getRunningServices(int maxNum)` *This method was deprecated in API level 26. As of* `Build.VERSION_CODES.O`, *this method is no longer available to third party applications. For backwards compatibility, it will still return the caller's own services.*
`List<ActivityManager.RunningTaskInfo>`	`getRunningTasks(int maxNum)` *This method was deprecated in API level 21. As of* `Build.VERSION_CODES.LOLLIPOP`, *this method is no longer available to third party applications: the introduction of document-centric recents means it can leak person information to the caller. For backwards compatibility, it will still return a small subset of its data: at least the caller's own tasks, and possibly some other tasks such as home that are known to not be sensitive.*
`boolean`	`isActivityStartAllowedOnDisplay(Context context, int displayId, Intent intent)` Check if the context is allowed to start an activity on specified display.
`boolean`	`isBackgroundRestricted()` Query whether the user has enabled background restrictions for this app.
`boolean`	`isInLockTaskMode()`

	This method was deprecated in API level 23. Use `getLockTaskModeState()` *instead.*
`static boolean`	`isLowMemoryKillReportSupported()`
`boolean`	`isLowRamDevice()` Returns true if this is a low-RAM device.
`static boolean`	`isRunningInTestHarness()` *This method was deprecated in API level 29. this method is false for all user builds. Users looking to check if their device is running in a device farm should see* `isRunningInUserTestHarness()` *.*
`static boolean`	`isRunningInUserTestHarness()` Returns "true" if the device is running in Test Harness Mode.
`static boolean`	`isUserAMonkey()` Returns "true" if the user interface is currently being messed with by a monkey.
`void`	`killBackgroundProcesses(String packageName)` Have the system immediately kill all background processes associated with the given package.
`void`	`moveTaskToFront(int taskId, int flags, Bundle options)` Ask that the task associated with a given task ID be moved to the front of the stack, so it is now visible to the user.
`void`	`moveTaskToFront(int taskId, int flags)` Equivalent to calling `moveTaskToFront(int,int,Bundle)` with a null options argument.
`void`	`registerAnrWarningListener(Executor executor, Consumer<AnrWarningResult> listener)` Registers a listener that is called when the app is close to the ANR timeout.
`void`	`removeApplicationStartInfoCompletionListener(Consumer<ApplicationStartInfo> listener)`

	Removes the provided callback set by `addApplicationStartInfoCompletionListener(Executor, Consumer)`.
`void`	`restartPackage(String packageName)` *This method was deprecated in API level 15. This is now just a wrapper for `killBackgroundProcesses(String)`; the previous behavior here is no longer available to applications because it allows them to break other applications by removing their alarms, stopping their services, etc.*
`void`	`setProcessStateSummary(byte[] state)` Set custom state data for this process.
`static void`	`setVrThread(int tid)` Enable more aggressive scheduling for latency-sensitive low-runtime VR threads.
`void`	`setWatchHeapLimit(long pssSize)` Request that the system start watching for the calling process to exceed a pss size as given here.
`void`	`unregisterAnrWarningListener(Consumer<AnrWarningResult> listener)` Unregisters a previously registered ANR warning listener.

Inherited methods

From class `java.lang.Object`

`Object`	`clone()` Creates and returns a copy of this object.
`boolean`	`equals(Object obj)` Indicates whether some other object is "equal to" this one.

`void`	`finalize()` Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
`final Class<?>`	`getClass()` Returns the runtime class of this `Object`.
`int`	`hashCode()` Returns a hash code value for the object.
`final void`	`notify()` Wakes up a single thread that is waiting on this object's monitor.
`final void`	`notifyAll()` Wakes up all threads that are waiting on this object's monitor.
`String`	`toString()` Returns a string representation of the object.
`final void`	`wait(long timeoutMillis, int nanos)` Causes the current thread to wait until it is awakened, typically by being *notified* or *interrupted*, or until a certain amount of real time has elapsed.
`final void`	`wait(long timeoutMillis)` Causes the current thread to wait until it is awakened, typically by being *notified* or *interrupted*, or until a certain amount of real time has elapsed.
`final void`	`wait()` Causes the current thread to wait until it is awakened, typically by being *notified* or *interrupted*.

Constants

ACTION_REPORT_HEAP_LIMIT

```
public static final String ACTION_REPORT_HEAP_LIMIT
```

Action an app can implement to handle reports from `setWatchHeapLimit(long)`. If your package has an activity handling this action, it will be launched with the heap data provided to it the same way as `Intent.ACTION_SEND`. Note that to match, the activity must support this action and a MIME type of "*/*".

Constant Value: "android.app.action.REPORT_HEAP_LIMIT"

LOCK_TASK_MODE_LOCKED

```
public static final int LOCK_TASK_MODE_LOCKED
```

Full lock task mode is active.

Constant Value: 1 (0x00000001)

LOCK_TASK_MODE_NONE

```
public static final int LOCK_TASK_MODE_NONE
```

Lock task mode is not active.

Constant Value: 0 (0x00000000)

LOCK_TASK_MODE_PINNED

```
public static final int LOCK_TASK_MODE_PINNED
```

App pinning mode is active.

Constant Value: 2 (0x00000002)

META_HOME_ALTERNATE

```
public static final String META_HOME_ALTERNATE
```

`<meta-data>` name for a 'home' Activity that declares a package that is to be uninstalled in lieu of the declaring one. The package named here must be signed with the same certificate as the one declaring the `<meta-data>`.

Constant Value: "android.app.home.alternate"

MOVE_TASK_NO_USER_ACTION

```
public static final int MOVE_TASK_NO_USER_ACTION
```

Flag for `moveTaskToFront(int, int)` : don't count this as a user-instigated action, so the current activity will not receive a hint that the user is leaving.

Constant Value: 2 (0x00000002)

RECENT_IGNORE_UNAVAILABLE

```
public static final int RECENT_IGNORE_UNAVAILABLE
```

Provides a list that does not contain any recent tasks that currently are not available to the user.

Constant Value: 2 (0x00000002)

Public methods

addAppTask

```
public int addAppTask (Activity activity,
                Intent intent,
                ActivityManager.TaskDescription description,
                Bitmap thumbnail)
```

Add a new `AppTask` for the calling application. This will create a new recents entry that is added to the **end** of all existing recents.

Parameters	
`activity`	`Activity` : The activity that is adding the entry. This is used to help determine the context that the new recents entry will be in. This value cannot be `null` .
`intent`	`Intent` : The Intent that describes the recents entry. This is the same Intent that you would have used to launch the activity for it. In generally you will want to set both `Intent.FLAG_ACTIVITY_NEW_DOCUMENT` and `Intent.FLAG_ACTIVITY_RETAIN_IN_RECENTS` ; the latter is required since this recents entry will exist without an activity, so it doesn't make sense to not retain it when its activity disappears. The given Intent here also must have an explicit ComponentName set on it. This value cannot be `null` .
`description`	`ActivityManager.TaskDescription` : Optional additional description information. This value may be `null` .

`thumbnail`	`Bitmap` : Thumbnail to use for the recents entry. Should be the size given by `getAppTaskThumbnailSize()` . If the bitmap is not that exact size, it will be recreated in your process, probably in a way you don't like, before the recents entry is added. This value cannot be `null` .
Returns	
`int`	Returns the task id of the newly added app task, or -1 if the add failed. The most likely cause of failure is that there is no more room for more tasks for your app.

addApplicationStartInfoCompletionListener

```
public void addApplicationStartInfoCompletionListener (Executor executor,
                Consumer<ApplicationStartInfo> listener)
```

Adds a callback that is notified when the `ApplicationStartInfo` record of this startup is complete. The startup is considered complete when the first frame is drawn. The callback doesn't wait for `Activity.reportFullyDrawn` to occur. Retrieve a copy of `ApplicationStartInfo` after `Activity.reportFullyDrawn` is called (using this callback or `ERROR(/getHistoricalProcessStartReasons)`) if you need the `ApplicationStartInfo.START_TIMESTAMP_FULLY_DRAWN` timestamp. If the current start record has already been completed (that is, the process is not currently starting), the callback will be invoked immediately on the specified executor with the previously completed `ApplicationStartInfo` record. Callback will be called at most once and removed automatically after being triggered.

Note: callback is asynchronous and should be made from a background thread.

Parameters	
`executor`	`Executor` : The executor on which the listener should be called. This value cannot be `null` .
`listener`	`Consumer` : Callback to be called when collection of `ApplicationStartInfo` is complete. Will replace existing listener if one is already attached. This value cannot be `null` .

addStartInfoTimestamp

```
public void addStartInfoTimestamp (int key,
                long timestampNs)
```

Adds an optional developer supplied timestamp to the calling apps most recent `ApplicationStartInfo` . This is in addition to system recorded timestamps.

Note: any timestamps added after `Activity.reportFullyDrawn` is called are discarded.

Note: will overwrite existing timestamp if called with same key.

appNotResponding

```
public void appNotResponding (String reason)
```

Method for the app to tell system that it's wedged and would like to trigger an ANR.

Parameters	
`reason`	`String` : The description of that what happened. This value cannot be `null` .

clearApplicationUserData

```
public boolean clearApplicationUserData ()
```

Permits an application to erase its own data from disk. This is equivalent to the user choosing to clear the app's data from within the device settings UI. It erases all dynamic data associated with the app -- its private data and data in its private area on external storage -- but does not remove the installed application itself, nor any OBB files. It also revokes all runtime permissions that the app has acquired, clears all notifications and removes all Uri grants related to this application.

Returns	
`boolean`	`true` if the application successfully requested that the application's data be erased; `false` otherwise.

clearWatchHeapLimit

```
public void clearWatchHeapLimit ()
```

Clear a heap watch limit previously set by `setWatchHeapLimit(long)` .

dumpPackageState

```
public void dumpPackageState (FileDescriptor fd,
                String packageName)
```

Perform a system dump of various state associated with the given application package name. This call blocks while the dump is being performed, so should not be done on a UI thread. The data will be written to the given file

descriptor as text.
Requires `Manifest.permission.DUMP`

Parameters	
`fd`	`FileDescriptor` : The file descriptor that the dump should be written to. The file descriptor is *not* closed by this function; the caller continues to own it.
`packageName`	`String` : The name of the package that is to be dumped.

getAppTasks

```
public List<ActivityManager.AppTask> getAppTasks ()
```

Get the list of tasks associated with the calling application.

Throws	
	java.lang.SecurityException

getDeviceConfigurationInfo

```
public ConfigurationInfo getDeviceConfigurationInfo ()
```

Get the device configuration attributes.

getHistoricalProcessExitReasons

```
public List<ApplicationExitInfo> getHistoricalProcessExitReasons (String packageName,
                int pid,
                int maxNum)
```

Return a list of `ApplicationExitInfo` records containing the reasons for the most recent app deaths.

Note: System stores this historical information in a ring buffer and only the most recent records will be returned.

Note: In the case that this application was bound to an external service with flag `Context.BIND_EXTERNAL_SERVICE` , the process of that external service will be included in this package's exit info.

Parameters	
`packageName`	`String` : Optional, a null value means match all packages belonging to the caller's UID. If this package belongs to another UID, you must hold `Manifest.permission.DUMP` in order to

	retrieve it.
`pid`	`int` : A process ID that used to belong to this package but died later; a value of 0 means to ignore this parameter and return all matching records. Value is 0 or greater
`maxNum`	`int` : The maximum number of results to be returned; a value of 0 means to ignore this parameter and return all matching records. Value is 0 or greater

getHistoricalProcessStartReasons

```
public List<ApplicationStartInfo> getHistoricalProcessStartReasons (int maxNum)
```

Return a list of `ApplicationStartInfo` records containing the information about the most recent app startups. Records accessed using this path might include "incomplete" records such as in-progress app starts. Accessing in-progress starts using this method lets you access start information early to better optimize your startup path.

Note: System stores this historical information in a ring buffer and only the most recent records will be returned.

Parameters	
`maxNum`	`int` : The maximum number of results to be returned; a value of 0 means to ignore this parameter and return all matching records. If fewer records exist, all existing records will be returned. Value is 0 or greater

getLargeMemoryClass

```
public int getLargeMemoryClass ()
```

Return the approximate per-application memory class of the current device when an application is running with a large heap. This is the space available for memory-intensive applications; most applications should not need this amount of memory, and should instead stay with the `getMemoryClass()` limit. The returned value is in megabytes. This may be the same size as `getMemoryClass()` on memory constrained devices, or it may be significantly larger on devices with a large amount of available RAM.

This is the size of the application's Dalvik heap if it has specified `android:largeHeap="true"` in its manifest.

getLauncherLargeIconDensity

```
public int getLauncherLargeIconDensity ()
```

Get the preferred density of icons for the launcher. This is used when custom drawables are created (e.g., for shortcuts).

Returns	
`int`	density in terms of DPI

getLauncherLargeIconSize

```
public int getLauncherLargeIconSize ()
```

Get the preferred launcher icon size. This is used when custom drawables are created (e.g., for shortcuts).

Returns	
`int`	dimensions of square icons in terms of pixels

getMemoryClass

```
public int getMemoryClass ()
```

Return the approximate per-application memory class of the current device. This gives you an idea of how hard a memory limit you should impose on your application to let the overall system work best. The returned value is in megabytes; the baseline Android memory class is 16 (which happens to be the Java heap limit of those devices); some devices with more memory may return 24 or even higher numbers.

getMemoryInfo

```
public void getMemoryInfo (ActivityManager.MemoryInfo outInfo)
```

Return general information about the memory state of the system. This can be used to help decide how to manage your own memory, though note that polling is not recommended and `ComponentCallbacks2.onTrimMemory(int)` is the preferred way to do this. Also see `getMyMemoryState(RunningAppProcessInfo)` for how to retrieve the current trim level of your process as needed, which gives a better hint for how to manage its memory.

Parameters	
`outInfo`	`ActivityManager.MemoryInfo`

getProcessMemoryInfo

```
public MemoryInfo[] getProcessMemoryInfo (int[] pids)
```

Return information about the memory usage of one or more processes.

Note: this method is only intended for debugging or building a user-facing process management UI.

As of `Android Q`, for regular apps this method will only return information about the memory info for the processes running as the caller's uid; no other process memory info is available and will be zero. Also of `Android Q` the sample rate allowed by this API is significantly limited, if called faster the limit you will receive the same data as the previous call.

Parameters	
`pids`	`int`: The pids of the processes whose memory usage is to be retrieved.
Returns	
`MemoryInfo[]`	Returns an array of memory information, one for each requested pid.

getRecentTasks

```
public List<ActivityManager.RecentTaskInfo> getRecentTasks (int maxNum,
                int flags)
```

This method was deprecated in API level 21.
As of `Build.VERSION_CODES.LOLLIPOP`, this method is no longer available to third party applications: the introduction of document-centric recents means it can leak personal information to the caller. For backwards compatibility, it will still return a small subset of its data: at least the caller's own tasks (though see `getAppTasks()` for the correct supported way to retrieve that information), and possibly some other tasks such as home that are known to not be sensitive.

Return a list of the tasks that the user has recently launched, with the most recent being first and older ones after in order.

Note: this method is only intended for debugging and presenting task management user interfaces. This should never be used for core logic in an application, such as deciding between different behaviors based on the information found here. Such uses are *not* supported, and will likely break in the future. For example, if multiple applications can be actively running at the same time, assumptions made about the meaning of the data here for purposes of control flow will be incorrect.

Parameters	
`maxNum`	`int`: The maximum number of entries to return in the list. The actual number returned may be smaller, depending on how many tasks the user has started and the maximum number the system can remember.

`flags`	`int` : Information about what to return. May be any combination of `RECENT_WITH_EXCLUDED` and `RECENT_IGNORE_UNAVAILABLE` .

getRunningAppProcesses

```
public List<ActivityManager.RunningAppProcessInfo> getRunningAppProcesses ()
```

Returns a list of application processes that are running on the device.

Note: this method is only intended for debugging or building a user-facing process management UI.

Returns	
`List<ActivityManager.RunningAppProcessInfo>`	Returns a list of RunningAppProcessInfo records, or null if there are no running processes (it will not return an empty list). This list ordering is not specified.

getRunningServiceControlPanel

```
public PendingIntent getRunningServiceControlPanel (ComponentName service)
```

Returns a PendingIntent you can start to show a control panel for the given running service. If the service does not have a control panel, null is returned.

Parameters	
`service`	`ComponentName`

getRunningServices

```
public List<ActivityManager.RunningServiceInfo> getRunningServices (int maxNum)
```

This method was deprecated in API level 26.
As of `Build.VERSION_CODES.O` , this method is no longer available to third party applications. For backwards compatibility, it will still return the caller's own services.

Return a list of the services that are currently running.

Note: this method is only intended for debugging or implementing service management type user interfaces.

Parameters

`maxNum`	`int` : The maximum number of entries to return in the list. The actual number returned may be smaller, depending on how many services are running.

getRunningTasks

```
public List<ActivityManager.RunningTaskInfo> getRunningTasks (int maxNum)
```

This method was deprecated in API level 21.
As of `Build.VERSION_CODES.LOLLIPOP` , this method is no longer available to third party applications: the introduction of document-centric recents means it can leak person information to the caller. For backwards compatibility, it will still return a small subset of its data: at least the caller's own tasks, and possibly some other tasks such as home that are known to not be sensitive.

Return a list of the tasks that are currently running, with the most recent being first and older ones after in order. Note that "running" does not mean any of the task's code is currently loaded or activity -- the task may have been frozen by the system, so that it can be restarted in its previous state when next brought to the foreground.

Note: this method is only intended for debugging and presenting task management user interfaces. This should never be used for core logic in an application, such as deciding between different behaviors based on the information found here. Such uses are *not* supported, and will likely break in the future. For example, if multiple applications can be actively running at the same time, assumptions made about the meaning of the data here for purposes of control flow will be incorrect.

Parameters	
`maxNum`	`int` : The maximum number of entries to return in the list. The actual number returned may be smaller, depending on how many tasks the user has started.

isActivityStartAllowedOnDisplay

```
public boolean isActivityStartAllowedOnDisplay (Context context,
                int displayId,
                Intent intent)
```

Check if the context is allowed to start an activity on specified display. Some launch restrictions may apply to secondary displays that are private, virtual, or owned by the system, in which case an activity start may throw a `SecurityException` . Call this method prior to starting an activity on a secondary display to check if the current context has access to it.

Parameters	
`context`	`Context` : Source context, from which an activity will be started. This value cannot be `null` .

`displayId`	`int` : Target display id.
`intent`	`Intent` : Intent used to launch an activity. This value cannot be `null` .
Returns	
`boolean`	`true` if a call to start an activity on the target display is allowed for the provided context and no `SecurityException` will be thrown, `false` otherwise.

isBackgroundRestricted

```
public boolean isBackgroundRestricted ()
```

Query whether the user has enabled background restrictions for this app.

The user may chose to do this, if they see that an app is consuming an unreasonable amount of battery while in the background.

If true, any work that the app tries to do will be aggressively restricted while it is in the background. At a minimum, jobs and alarms will not execute and foreground services cannot be started unless an app activity is in the foreground.

Note that these restrictions stay in effect even when the device is charging.

Returns	
`boolean`	true if user has enforced background restrictions for this app, false otherwise.

isInLockTaskMode

```
public boolean isInLockTaskMode ()
```

This method was deprecated in API level 23.
Use `getLockTaskModeState()` instead.

Return whether currently in lock task mode. When in this mode no new tasks can be created or switched to.

isLowMemoryKillReportSupported

```
public static boolean isLowMemoryKillReportSupported ()
```

isLowRamDevice

```
public boolean isLowRamDevice ()
```

Returns true if this is a low-RAM device. Exactly whether a device is low-RAM is ultimately up to the device configuration, but currently it generally means something with 1GB or less of RAM. This is mostly intended to be used by apps to determine whether they should turn off certain features that require more RAM.

isRunningInTestHarness

```
public static boolean isRunningInTestHarness ()
```

This method was deprecated in API level 29.
this method is false for all user builds. Users looking to check if their device is running in a device farm should see `isRunningInUserTestHarness()`.

Returns "true" if device is running in a test harness.

isRunningInUserTestHarness

```
public static boolean isRunningInUserTestHarness ()
```

Returns "true" if the device is running in Test Harness Mode.

Test Harness Mode is a feature that allows devices to run without human interaction in a device farm/testing harness (such as Firebase Test Lab). You should check this method if you want your app to behave differently when running in a test harness to skip setup screens that would impede UI testing. e.g. a keyboard application that has a full screen setup page for the first time it is launched.

Note that you should *not* use this to determine whether or not your app is running an instrumentation test, as it is not set for a standard device running a test.

isUserAMonkey

```
public static boolean isUserAMonkey ()
```

Returns "true" if the user interface is currently being messed with by a monkey.

killBackgroundProcesses

```
public void killBackgroundProcesses (String packageName)
```

Have the system immediately kill all background processes associated with the given package. This is the same as the kernel killing those processes to reclaim memory; the system will take care of restarting these processes in the

future as needed.

On devices that run Android 14 or higher, third party applications can only use this API to kill their own processes.

.
Requires `Manifest.permission.KILL_BACKGROUND_PROCESSES`

Parameters	
`packageName`	`String` : The name of the package whose processes are to be killed.

moveTaskToFront

```
public void moveTaskToFront (int taskId,
                int flags,
                Bundle options)
```

Ask that the task associated with a given task ID be moved to the front of the stack, so it is now visible to the user.
Requires `Manifest.permission.REORDER_TASKS`

registerAnrWarningListener

```
public void registerAnrWarningListener (Executor executor,
                Consumer<AnrWarningResult> listener)
```

Registers a listener that is called when the app is close to the ANR timeout.

This is intended to give the app a chance to collect and store any additional information they may want to gather at this time, or take any pre-ANR actions. Note that these listeners are called at best-effort, and may not be successfully called (or be provided time to execute) before the ANR occurs and the app is killed.

If the app registers multiple distinct listeners, all registered listeners will be notified on the potential ANR condition. The order in which listeners are notified is not guaranteed.

The app can unregister the listener using `unregisterAnrWarningListener(Consumer)` .

Parameters	
`executor`	`Executor` : The executor on which listener will be invoked. This should not be the application's main thread. This value cannot be `null` .
`listener`	`Consumer` : The listener to be triggered on the ANR warning condition. This value cannot be `null` .

restartPackage

```
public void restartPackage (String packageName)
```

This method was deprecated in API level 15.
This is now just a wrapper for `killBackgroundProcesses(String)` ; the previous behavior here is no longer available to applications because it allows them to break other applications by removing their alarms, stopping their services, etc.

Parameters	
`packageName`	`String`

setProcessStateSummary

```
public void setProcessStateSummary (byte[] state)
```

Set custom state data for this process. It will be included in the record of `ApplicationExitInfo` on the death of the current calling process; the new process of the app can retrieve this state data by calling `ApplicationExitInfo.getProcessStateSummary()` on the record returned by `getHistoricalProcessExitReasons(String, int, int)` .

This would be useful for the calling app to save its stateful data: if it's killed later for any reason, the new process of the app can know what the previous process of the app was doing. For instance, you could use this to encode the current level in a game, or a set of features/experiments that were enabled. Later you could analyze under what circumstances the app tends to crash or use too much memory. However, it's not suggested to rely on this to restore the applications previous UI state or so, it's only meant for analyzing application healthy status.

System might decide to throttle the calls to this API; so call this API in a reasonable manner, excessive calls to this API could result a `RuntimeException` .

Parameters	
`state`	`byte` : The state data. To be advised, **DO NOT** include sensitive information/data (PII, SPII, or other sensitive user data) here. Maximum length is 128 bytes. This value may be `null` .

setVrThread

```
public static void setVrThread (int tid)
```

Enable more aggressive scheduling for latency-sensitive low-runtime VR threads. Only one thread can be a VR thread in a process at a time, and that thread may be subject to restrictions on the amount of time it can run. If

persistent VR mode is set, whatever thread has been granted aggressive scheduling via this method will return to normal operation, and calling this method will do nothing while persistent VR mode is enabled. To reset the VR thread for an application, a tid of 0 can be passed.

Parameters	
`tid`	`int` : tid of the VR thread

setWatchHeapLimit

```
public void setWatchHeapLimit (long pssSize)
```

Request that the system start watching for the calling process to exceed a pss size as given here. Once called, the system will look for any occasions where it sees the associated process with a larger pss size and, when this happens, automatically pull a heap dump from it and allow the user to share the data. Note that this request continues running even if the process is killed and restarted. To remove the watch, use `clearWatchHeapLimit()` .

This API only works if the calling process has been marked as `ApplicationInfo.FLAG_DEBUGGABLE` or this is running on a debuggable (userdebug or eng) build.

Callers can optionally implement `ACTION_REPORT_HEAP_LIMIT` to directly handle heap limit reports themselves.

Parameters	
`pssSize`	`long` : The size in bytes to set the limit at.

unregisterAnrWarningListener

```
public void unregisterAnrWarningListener (Consumer<AnrWarningResult> listener)
```

Unregisters a previously registered ANR warning listener.

Source: https://developer.android.com/reference/android/app/ActivityManager.html#getRunningTasks%28int%29