DevicePolicyManager | API reference | Android Developers

Archived: 2026-04-05 21:29:52 UTC

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

```
public class DevicePolicyManager
extends Object
```

Manages device policy and restrictions applied to the user of the device or apps running on the device.

This class contains three types of methods:

1. Those aimed at managing apps
2. Those aimed at the Device Policy Management Role Holder
3. Those aimed at apps which wish to respect device policy

The intended caller for each API is indicated in its Javadoc.

Managing Apps

Apps can be made capable of setting device policy ("Managing Apps") either by being set as a Device Administrator, being set as a Device Policy Controller, or by holding the appropriate Permissions.

A **Device Administrator** is an app which is able to enforce device policies that it has declared in its device admin XML file. An app can prompt the user to give it device administrator privileges using the `ACTION_ADD_DEVICE_ADMIN` action.

For more information about Device Administration, read the Device Administration developer guide.

Device Administrator apps can also be recognised as **Device Policy Controllers**. Device Policy Controllers can be one of two types:

- A *Device Owner*, which only ever exists on the `System User` or Main User, is the most powerful type of Device Policy Controller and can affect policy across the device.
- A *Profile Owner*, which can exist on any user, can affect policy on the user it is on, and when it is running on `a profile` has limited ability to affect policy on its parent.

Additional capabilities can be provided to Device Policy Controllers in the following circumstances:

- A Profile Owner on an organization owned device has access to additional abilities, both affecting policy on the profile's parent and also the profile itself.
- A Profile Owner running on the `System User` has access to additional capabilities which affect the `System User` and also the whole device.

- A Profile Owner running on an affiliated user has capabilities similar to that of a Device Owner

For more information, see Building a Device Policy Controller.

Permissions are generally only given to apps fulfilling particular key roles on the device (such as managing `device locks`).

Device Policy Management Role Holder

One app on the device fulfills the Device Policy Management Role and is trusted with managing the overall state of Device Policy. This has access to much more powerful methods than managing apps.

Querying Device Policy

In most cases, regular apps do not need to concern themselves with device policy, and restrictions will be enforced automatically. There are some cases where an app may wish to query device policy to provide a better user experience. Only a small number of policies allow apps to query them directly. These APIs will typically have no special required permissions.

Managed Provisioning

Managed Provisioning is the process of recognising an app as a Device Owner or Profile Owner. It involves presenting education and consent screens to the user to ensure they are aware of the capabilities this grants the Device Policy Controller

For more information on provisioning, see Building a Device Policy Controller.

A **Managed Profile** enables data separation. For example to use a device both for personal and corporate usage. The managed profile and its parent share a launcher.

Affiliation

Using the `setAffiliationIds(ComponentName, Set)` method, a Device Owner can set a list of affiliation ids for the `System User`. Any Profile Owner on the same device can also call `setAffiliationIds(ComponentName, Set)` to set affiliation ids for the `user` it is on. When there is the same ID present in both lists, the user is said to be "affiliated" and we can refer to the Profile Owner as a "profile owner on an affiliated user" or an "affiliated profile owner". Becoming affiliated grants the Profile Owner capabilities similar to that of the Device Owner. It also allows use of the `bindDeviceAdminServiceAsUser(ComponentName, Intent, ServiceConnection, BindServiceFlags, UserHandle)` APIs for direct communication between the Device Owner and affiliated Profile Owners.

Organization Owned

An organization owned device is one which is not owned by the person making use of the device and is instead owned by an organization such as their employer or education provider. These devices are recognised as being organization owned either by the presence of a device owner or of a `profile which has a profile owner is marked as organization owned`.

Profile owners running on an organization owned device can exercise additional capabilities using the `getParentProfileInstance(ComponentName)` API which apply to the parent user. Each API will indicate if it is usable in this way.

Android Automotive

On `"Android Automotive builds"`, some methods can throw `"an exception"` if an action is unsafe (for example, if the vehicle is moving). Callers running on `"Android Automotive builds"` should always check for this exception.
Restricted for SDK Runtime environment in API level 34.
Requires the `PackageManager#FEATURE_DEVICE_ADMIN` feature which can be detected using `PackageManager.hasSystemFeature(String)`.

Summary

Nested classes	
`class`	`DevicePolicyManager.InstallSystemUpdateCallback` Callback used in `DevicePolicyManager.installSystemUpdate(ComponentName, Uri, Executor, InstallSystemUpdateCallback)` to indicate that there was an error while trying to install an update.
`interface`	`DevicePolicyManager.OnClearApplicationUserDataListener` Callback used in `DevicePolicyManager.clearApplicationUserData(ComponentName, String, Executor, OnClearApplicationUserDataListener)` to indicate that the clearing of an application's user data is done.

Constants	
`String`	`ACTION_ADD_DEVICE_ADMIN` Activity action: ask the user to add a new device administrator to the system.
`String`	`ACTION_ADMIN_POLICY_COMPLIANCE` Activity action: Starts the administrator to show policy compliance for the provisioning.
`String`	`ACTION_APPLICATION_DELEGATION_SCOPES_CHANGED` Broadcast Action: Sent after application delegation scopes are changed.
`String`	`ACTION_CHECK_POLICY_COMPLIANCE` Activity action: launch the DPC to check policy compliance.

String	ACTION_DEVICE_ADMIN_SERVICE Service action: Action for a service that device owner and profile owner can optionally own.
String	ACTION_DEVICE_FINANCING_STATE_CHANGED Broadcast Action: Broadcast sent to indicate that the device financing state has changed.
String	ACTION_DEVICE_OWNER_CHANGED Broadcast action: sent when the device owner is set, changed or cleared.
String	ACTION_DEVICE_POLICY_RESOURCE_UPDATED Broadcast action: notify system apps (e.g. settings, SysUI, etc) that the device management resources with IDs EXTRA_RESOURCE_IDS has been updated, the updated resources can be retrieved using DevicePolicyResourcesManager.getDrawable and DevicePolicyResourcesManager.getString.
String	ACTION_GET_PROVISIONING_MODE Activity action: Starts the administrator to get the mode for the provisioning.
String	ACTION_MANAGED_PROFILE_PROVISIONED Broadcast Action: This broadcast is sent to indicate that provisioning of a managed profile has completed successfully.
String	ACTION_PROFILE_OWNER_CHANGED Broadcast action: sent when the profile owner is set, changed or cleared.
String	ACTION_PROVISIONING_SUCCESSFUL Activity action: This activity action is sent to indicate that provisioning of a managed profile or managed device has completed successfully.
String	ACTION_PROVISION_MANAGED_DEVICE *This constant was deprecated in API level 31. to support Build.VERSION_CODES.S and later, admin apps must implement activities with intent filters for the ACTION_GET_PROVISIONING_MODE and ACTION_ADMIN_POLICY_COMPLIANCE intent actions; using ACTION_PROVISION_MANAGED_DEVICE to start provisioning will cause the provisioning to fail; to additionally support pre- Build.VERSION_CODES.S, admin apps must also continue to use this constant.*
String	ACTION_PROVISION_MANAGED_PROFILE

	Activity action: Starts the provisioning flow which sets up a managed profile.
String	ACTION_SET_NEW_PARENT_PROFILE_PASSWORD Activity action: have the user enter a new password for the parent profile.
String	ACTION_SET_NEW_PASSWORD Activity action: have the user enter a new password.
String	ACTION_START_ENCRYPTION Activity action: begin the process of encrypting data on the device.
String	ACTION_SYSTEM_UPDATE_POLICY_CHANGED Broadcast action: notify that a new local system update policy has been set by the device owner.
int	APP_FUNCTIONS_DISABLED Indicates that AppFunctionManager is controlled and disabled by policy, i.e.
int	APP_FUNCTIONS_DISABLED_CROSS_PROFILE Indicates that AppFunctionManager is controlled and disabled by a policy for cross profile interactions only, i.e.
int	APP_FUNCTIONS_NOT_CONTROLLED_BY_POLICY Indicates that AppFunctionManager is not controlled by policy.
int	AUTO_TIME_DISABLED Specifies the "disabled" auto time state.
int	AUTO_TIME_ENABLED Specifies the "enabled" auto time state.
int	AUTO_TIME_NOT_CONTROLLED_BY_POLICY Specifies that the auto time state is not controlled by device policy.

int	AUTO_TIME_ZONE_DISABLED Specifies the "disabled" auto time zone state.
int	AUTO_TIME_ZONE_ENABLED Specifies the "enabled" auto time zone state.
int	AUTO_TIME_ZONE_NOT_CONTROLLED_BY_POLICY Specifies that the auto time zone state is not controlled by device policy.
int	CONTENT_PROTECTION_DISABLED Indicates that content protection is controlled and disabled by a policy (default).
int	CONTENT_PROTECTION_ENABLED Indicates that content protection is controlled and enabled by a policy.
int	CONTENT_PROTECTION_NOT_CONTROLLED_BY_POLICY Indicates that content protection is not controlled by policy, allowing user to choose.
String	DELEGATION_APP_RESTRICTIONS Delegation of application restrictions management.
String	DELEGATION_BLOCK_UNINSTALL Delegation of application uninstall block.
String	DELEGATION_CERT_INSTALL Delegation of certificate installation and management.
String	DELEGATION_CERT_SELECTION Grants access to selection of KeyChain certificates on behalf of requesting apps.
String	DELEGATION_ENABLE_SYSTEM_APP Delegation for enabling system apps.
String	DELEGATION_INSTALL_EXISTING_PACKAGE

	Delegation for installing existing packages.
String	DELEGATION_KEEP_UNINSTALLED_PACKAGES Delegation of management of uninstalled packages.
String	DELEGATION_NETWORK_LOGGING Grants access to setNetworkLoggingEnabled(ComponentName, boolean), isNetworkLoggingEnabled(ComponentName) and retrieveNetworkLogs(ComponentName, long).
String	DELEGATION_PACKAGE_ACCESS Delegation of package access state.
String	DELEGATION_PERMISSION_GRANT Delegation of permission policy and permission grant state.
String	DELEGATION_SECURITY_LOGGING Grants access to setSecurityLoggingEnabled(ComponentName, boolean), isSecurityLoggingEnabled(ComponentName), retrieveSecurityLogs(ComponentName), and retrievePreRebootSecurityLogs(ComponentName).
int	ENCRYPTION_STATUS_ACTIVATING *This constant was deprecated in API level 34. This result code has never actually been used, so there is no reason for apps to check for it.*
int	ENCRYPTION_STATUS_ACTIVE Result code for setStorageEncryption(ComponentName, boolean) and getStorageEncryptionStatus(): indicating that encryption is active.
int	ENCRYPTION_STATUS_ACTIVE_DEFAULT_KEY Result code for getStorageEncryptionStatus(): indicating that encryption is active, but the encryption key is not cryptographically protected by the user's credentials.
int	ENCRYPTION_STATUS_ACTIVE_PER_USER Result code for getStorageEncryptionStatus(): indicating that encryption is active and the encryption key is tied to the user or profile.

int	`ENCRYPTION_STATUS_INACTIVE` Result code for `setStorageEncryption(ComponentName, boolean)` and `getStorageEncryptionStatus()` : indicating that encryption is supported, but is not currently active.
int	`ENCRYPTION_STATUS_UNSUPPORTED` Result code for `setStorageEncryption(ComponentName, boolean)` and `getStorageEncryptionStatus()` : indicating that encryption is not supported.
`String`	`EXTRA_ADD_EXPLANATION` An optional CharSequence providing additional explanation for why the admin is being added.
`String`	`EXTRA_DELEGATION_SCOPES` An `ArrayList<String>` corresponding to the delegation scopes given to an app in the `ACTION_APPLICATION_DELEGATION_SCOPES_CHANGED` broadcast.
`String`	`EXTRA_DEVICE_ADMIN` The ComponentName of the administrator component.
`String`	`EXTRA_DEVICE_PASSWORD_REQUIREMENT_ONLY` A boolean extra for `ACTION_SET_NEW_PARENT_PROFILE_PASSWORD` requesting that only device password requirement is enforced during the parent profile password enrolment flow.
`String`	`EXTRA_PASSWORD_COMPLEXITY` An integer indicating the complexity level of the new password an app would like the user to set when launching the action `ACTION_SET_NEW_PASSWORD` .
`String`	`EXTRA_PROVISIONING_ACCOUNT_TO_MIGRATE` An `Account` extra holding the account to migrate during managed profile provisioning.
`String`	`EXTRA_PROVISIONING_ADMIN_EXTRAS_BUNDLE` A `Parcelable` extra of type `PersistableBundle` that is passed directly to the Device Policy Controller after provisioning.
`String`	`EXTRA_PROVISIONING_ALLOWED_PROVISIONING_MODES` An `ArrayList` of `Integer` extra specifying the allowed provisioning modes.

String	EXTRA_PROVISIONING_ALLOW_OFFLINE A boolean extra indicating whether offline provisioning should be used.
String	EXTRA_PROVISIONING_DEVICE_ADMIN_COMPONENT_NAME A ComponentName extra indicating the device admin receiver of the application that will be set as the Device Policy Controller.
String	EXTRA_PROVISIONING_DEVICE_ADMIN_MINIMUM_VERSION_CODE An int extra holding a minimum required version code for the device admin package.
String	EXTRA_PROVISIONING_DEVICE_ADMIN_PACKAGE_CHECKSUM A String extra holding the URL-safe base64 encoded SHA-256 hash of the file at download location specified in EXTRA_PROVISIONING_DEVICE_ADMIN_PACKAGE_DOWNLOAD_LOCATION.
String	EXTRA_PROVISIONING_DEVICE_ADMIN_PACKAGE_DOWNLOAD_COOKIE_HEADER A String extra holding a http cookie header which should be used in the http request to the url specified in EXTRA_PROVISIONING_DEVICE_ADMIN_PACKAGE_DOWNLOAD_LOCATION.
String	EXTRA_PROVISIONING_DEVICE_ADMIN_PACKAGE_DOWNLOAD_LOCATION A String extra holding a url that specifies the download location of the device admin package.
String	EXTRA_PROVISIONING_DEVICE_ADMIN_PACKAGE_NAME *This constant was deprecated in API level 23. Use EXTRA_PROVISIONING_DEVICE_ADMIN_COMPONENT_NAME.*
String	EXTRA_PROVISIONING_DEVICE_ADMIN_SIGNATURE_CHECKSUM A String extra holding the URL-safe base64 encoded SHA-256 checksum of any signature of the android package archive at the download location specified in EXTRA_PROVISIONING_DEVICE_ADMIN_PACKAGE_DOWNLOAD_LOCATION.
String	EXTRA_PROVISIONING_DISCLAIMERS A Bundle [] extra consisting of list of disclaimer headers and disclaimer contents.
String	EXTRA_PROVISIONING_DISCLAIMER_CONTENT A Uri extra pointing to disclaimer content.

String	EXTRA_PROVISIONING_DISCLAIMER_HEADER A String extra of localized disclaimer header.
String	EXTRA_PROVISIONING_EMAIL_ADDRESS *This constant was deprecated in API level 26. From Build.VERSION_CODES.O , never used while provisioning the device.*
String	EXTRA_PROVISIONING_IMEI A string extra holding the IMEI (International Mobile Equipment Identity) of the device.
String	EXTRA_PROVISIONING_KEEP_ACCOUNT_ON_MIGRATION Boolean extra to indicate that the migrated_account should be kept.
String	EXTRA_PROVISIONING_KEEP_SCREEN_ON *This constant was deprecated in API level 34. from Build.VERSION_CODES.UPSIDE_DOWN_CAKE , the flag wouldn't be functional. The screen is kept on throughout the provisioning flow.*
String	EXTRA_PROVISIONING_LEAVE_ALL_SYSTEM_APPS_ENABLED A Boolean extra that can be used by the mobile device management application to skip the disabling of system apps during provisioning when set to true .
String	EXTRA_PROVISIONING_LOCALE A String extra holding the Locale that the device will be set to.
String	EXTRA_PROVISIONING_LOCAL_TIME A Long extra holding the wall clock time (in milliseconds) to be set on the device's AlarmManager .
String	EXTRA_PROVISIONING_LOGO_URI *This constant was deprecated in API level 33. Logo customization is no longer supported in the provisioning flow.*
String	EXTRA_PROVISIONING_MAIN_COLOR *This constant was deprecated in API level 31. Color customization is no longer supported in the provisioning flow.*

String	EXTRA_PROVISIONING_MODE An intent extra holding the provisioning mode returned by the administrator.
String	EXTRA_PROVISIONING_SENSORS_PERMISSION_GRANT_OPT_OUT A boolean extra indicating the admin of a fully-managed device opts out of controlling permission grants for sensor-related permissions, see setPermissionGrantState(ComponentName,String,String,int).
String	EXTRA_PROVISIONING_SERIAL_NUMBER A string extra holding the serial number of the device.
String	EXTRA_PROVISIONING_SHOULD_LAUNCH_RESULT_INTENT A boolean extra that determines whether the provisioning flow should launch the resulting launch intent, if one is supplied by the device policy management role holder via EXTRA_RESULT_LAUNCH_INTENT.
String	EXTRA_PROVISIONING_SKIP_EDUCATION_SCREENS A boolean extra indicating if the education screens from the provisioning flow should be skipped.
String	EXTRA_PROVISIONING_SKIP_ENCRYPTION A boolean extra indicating whether device encryption can be skipped as part of provisioning.
String	EXTRA_PROVISIONING_SKIP_USER_CONSENT *This constant was deprecated in API level 31. this extra is no longer relevant as device owners cannot create managed profiles*
String	EXTRA_PROVISIONING_TIME_ZONE A String extra holding the time zone AlarmManager that the device will be set to.
String	EXTRA_PROVISIONING_USE_MOBILE_DATA A boolean extra indicating if mobile data should be used during the provisioning flow for downloading the admin app.
String	EXTRA_PROVISIONING_WIFI_ANONYMOUS_IDENTITY The anonymous identity of the wifi network in EXTRA_PROVISIONING_WIFI_SSID.

String	EXTRA_PROVISIONING_WIFI_CA_CERTIFICATE The CA certificate of the wifi network in EXTRA_PROVISIONING_WIFI_SSID.
String	EXTRA_PROVISIONING_WIFI_DOMAIN The domain of the wifi network in EXTRA_PROVISIONING_WIFI_SSID.
String	EXTRA_PROVISIONING_WIFI_EAP_METHOD The EAP method of the wifi network in EXTRA_PROVISIONING_WIFI_SSID and could be one of PEAP, TLS, TTLS, PWD, SIM, AKA or AKA_PRIME.
String	EXTRA_PROVISIONING_WIFI_HIDDEN A boolean extra indicating whether the wifi network in EXTRA_PROVISIONING_WIFI_SSID is hidden or not.
String	EXTRA_PROVISIONING_WIFI_IDENTITY The identity of the wifi network in EXTRA_PROVISIONING_WIFI_SSID.
String	EXTRA_PROVISIONING_WIFI_PAC_URL A String extra holding the proxy auto-config (PAC) URL for the wifi network in EXTRA_PROVISIONING_WIFI_SSID.
String	EXTRA_PROVISIONING_WIFI_PASSWORD A String extra holding the password of the wifi network in EXTRA_PROVISIONING_WIFI_SSID.
String	EXTRA_PROVISIONING_WIFI_PHASE2_AUTH The phase 2 authentication of the wifi network in EXTRA_PROVISIONING_WIFI_SSID and could be one of NONE, PAP, MSCHAP, MSCHAPV2, GTC, SIM, AKA or AKA_PRIME.
String	EXTRA_PROVISIONING_WIFI_PROXY_BYPASS A String extra holding the proxy bypass for the wifi network in EXTRA_PROVISIONING_WIFI_SSID.
String	EXTRA_PROVISIONING_WIFI_PROXY_HOST A String extra holding the proxy host for the wifi network in EXTRA_PROVISIONING_WIFI_SSID.
String	EXTRA_PROVISIONING_WIFI_PROXY_PORT

	An int extra holding the proxy port for the wifi network in `EXTRA_PROVISIONING_WIFI_SSID`.
`String`	`EXTRA_PROVISIONING_WIFI_SECURITY_TYPE` A String extra indicating the security type of the wifi network in `EXTRA_PROVISIONING_WIFI_SSID` and could be one of `NONE`, `WPA`, `WEP` or `EAP`.
`String`	`EXTRA_PROVISIONING_WIFI_SSID` A String extra holding the ssid of the wifi network that should be used during nfc device owner provisioning for downloading the mobile device management application.
`String`	`EXTRA_PROVISIONING_WIFI_USER_CERTIFICATE` The user certificate of the wifi network in `EXTRA_PROVISIONING_WIFI_SSID`.
`String`	`EXTRA_RESOURCE_IDS` An integer array extra for `ACTION_DEVICE_POLICY_RESOURCE_UPDATED` to indicate which resource IDs (i.e.
`String`	`EXTRA_RESOURCE_TYPE` An `int` extra for `ACTION_DEVICE_POLICY_RESOURCE_UPDATED` to indicate the type of the resource being updated, the type can be `EXTRA_RESOURCE_TYPE_DRAWABLE` or `EXTRA_RESOURCE_TYPE_STRING`
`int`	`EXTRA_RESOURCE_TYPE_DRAWABLE` A `int` value for `EXTRA_RESOURCE_TYPE` to indicate that a resource of type `Drawable` is being updated.
`int`	`EXTRA_RESOURCE_TYPE_STRING` A `int` value for `EXTRA_RESOURCE_TYPE` to indicate that a resource of type `String` is being updated.
`String`	`EXTRA_RESULT_LAUNCH_INTENT` An `Intent` result extra specifying the `Intent` to be launched after provisioning is finalized.
`int`	`FLAG_EVICT_CREDENTIAL_ENCRYPTION_KEY` Flag for `lockNow(int)`: also evict the user's credential encryption key from the keyring.
`int`	`FLAG_MANAGED_CAN_ACCESS_PARENT`

	Flag used by `addCrossProfileIntentFilter(ComponentName, IntentFilter, int)` to allow activities in the managed profile to access intents sent from the parent profile.
`int`	`FLAG_PARENT_CAN_ACCESS_MANAGED` Flag used by `addCrossProfileIntentFilter(ComponentName, IntentFilter, int)` to allow activities in the parent profile to access intents sent from the managed profile.
`int`	`ID_TYPE_BASE_INFO` Specifies that the device should attest its manufacturer details.
`int`	`ID_TYPE_IMEI` Specifies that the device should attest its IMEI.
`int`	`ID_TYPE_INDIVIDUAL_ATTESTATION` Specifies that the device should attest using an individual attestation certificate.
`int`	`ID_TYPE_MEID` Specifies that the device should attest its MEID.
`int`	`ID_TYPE_SERIAL` Specifies that the device should attest its serial number.
`int`	`INSTALLKEY_REQUEST_CREDENTIALS_ACCESS` Specifies that the calling app should be granted access to the installed credentials immediately.
`int`	`INSTALLKEY_SET_USER_SELECTABLE` Specifies that a user can select the key via the Certificate Selection prompt.
`int`	`KEYGUARD_DISABLE_BIOMETRICS` Disable all biometric authentication on keyguard secure screens (e.g. PIN/Pattern/Password).
`int`	`KEYGUARD_DISABLE_FACE` Disable face authentication on keyguard secure screens (e.g. PIN/Pattern/Password).
`int`	`KEYGUARD_DISABLE_FEATURES_ALL`

	Disable all current and future keyguard customizations.
int	`KEYGUARD_DISABLE_FEATURES_NONE` Widgets are enabled in keyguard
int	`KEYGUARD_DISABLE_FINGERPRINT` Disable fingerprint authentication on keyguard secure screens (e.g. PIN/Pattern/Password).
int	`KEYGUARD_DISABLE_IRIS` Disable iris authentication on keyguard secure screens (e.g. PIN/Pattern/Password).
int	`KEYGUARD_DISABLE_REMOTE_INPUT` *This constant was deprecated in API level 33. This flag was added in version* `Build.VERSION_CODES.N` *, but it never had any effect.*
int	`KEYGUARD_DISABLE_SECURE_CAMERA` Disable the camera on secure keyguard screens (e.g. PIN/Pattern/Password)
int	`KEYGUARD_DISABLE_SECURE_NOTIFICATIONS` Disable showing all notifications on secure keyguard screens (e.g. PIN/Pattern/Password)
int	`KEYGUARD_DISABLE_SHORTCUTS_ALL` Disable all keyguard shortcuts.
int	`KEYGUARD_DISABLE_TRUST_AGENTS` Disable trust agents on secure keyguard screens (e.g. PIN/Pattern/Password).
int	`KEYGUARD_DISABLE_UNREDACTED_NOTIFICATIONS` Only allow redacted notifications on secure keyguard screens (e.g. PIN/Pattern/Password)
int	`KEYGUARD_DISABLE_WIDGETS_ALL` Disable all keyguard widgets.
int	`LEAVE_ALL_SYSTEM_APPS_ENABLED`

	Flag used by `createAndManageUser(ComponentName, String, ComponentName, PersistableBundle, int)` to specify that the newly created user should skip the disabling of system apps during provisioning.
`int`	`LOCK_TASK_FEATURE_BLOCK_ACTIVITY_START_IN_TASK` Enable blocking of non-allowlisted activities from being started into a locked task.
`int`	`LOCK_TASK_FEATURE_GLOBAL_ACTIONS` Enable the global actions dialog during LockTask mode.
`int`	`LOCK_TASK_FEATURE_HOME` Enable the Home button during LockTask mode.
`int`	`LOCK_TASK_FEATURE_KEYGUARD` Enable the keyguard during LockTask mode.
`int`	`LOCK_TASK_FEATURE_NONE` Disable all configurable SystemUI features during LockTask mode.
`int`	`LOCK_TASK_FEATURE_NOTIFICATIONS` Enable notifications during LockTask mode.
`int`	`LOCK_TASK_FEATURE_OVERVIEW` Enable the Overview button and the Overview screen during LockTask mode.
`int`	`LOCK_TASK_FEATURE_SYSTEM_INFO` Enable the system info area in the status bar during LockTask mode.
`int`	`MAKE_USER_EPHEMERAL` Flag used by `createAndManageUser(ComponentName, String, ComponentName, PersistableBundle, int)` to specify that the user should be created ephemeral.
`String`	`MIME_TYPE_PROVISIONING_NFC` This MIME type is used for starting the device owner provisioning.

`int`	`MTE_DISABLED` Require that MTE be disabled on the device.
`int`	`MTE_ENABLED` Require that MTE be enabled on the device, if supported.
`int`	`MTE_NOT_CONTROLLED_BY_POLICY` Allow the user to choose whether to enable MTE on the device.
`int`	`NEARBY_STREAMING_DISABLED` Indicates that nearby streaming is disabled.
`int`	`NEARBY_STREAMING_ENABLED` Indicates that nearby streaming is enabled.
`int`	`NEARBY_STREAMING_NOT_CONTROLLED_BY_POLICY` Indicates that nearby streaming is not controlled by policy, which means nearby streaming is allowed.
`int`	`NEARBY_STREAMING_SAME_MANAGED_ACCOUNT_ONLY` Indicates that nearby streaming is enabled only to devices offering a comparable level of security, with the same authenticated managed account.
`int`	`OPERATION_SAFETY_REASON_DRIVING_DISTRACTION` Indicates that a `UnsafeStateException` was thrown because the operation would distract the driver of the vehicle.
`int`	`PASSWORD_COMPLEXITY_HIGH` Constant for `getPasswordComplexity()` and `setRequiredPasswordComplexity(int)`.
`int`	`PASSWORD_COMPLEXITY_LOW` Constant for `getPasswordComplexity()` and `setRequiredPasswordComplexity(int)`.
`int`	`PASSWORD_COMPLEXITY_MEDIUM` Constant for `getPasswordComplexity()` and `setRequiredPasswordComplexity(int)`.

`int`	`PASSWORD_COMPLEXITY_NONE` Constant for `getPasswordComplexity()` and `setRequiredPasswordComplexity(int)` : no password.
`int`	`PASSWORD_QUALITY_ALPHABETIC` Constant for `setPasswordQuality(ComponentName, int)` : the user must have entered a password containing at least alphabetic (or other symbol) characters.
`int`	`PASSWORD_QUALITY_ALPHANUMERIC` Constant for `setPasswordQuality(ComponentName, int)` : the user must have entered a password containing at least *both*> numeric *and* alphabetic (or other symbol) characters.
`int`	`PASSWORD_QUALITY_BIOMETRIC_WEAK` Constant for `setPasswordQuality(ComponentName, int)` : the policy allows for low-security biometric recognition technology.
`int`	`PASSWORD_QUALITY_COMPLEX` Constant for `setPasswordQuality(ComponentName, int)` : allows the admin to set precisely how many characters of various types the password should contain to satisfy the policy.
`int`	`PASSWORD_QUALITY_NUMERIC` Constant for `setPasswordQuality(ComponentName, int)` : the user must have entered a password containing at least numeric characters.
`int`	`PASSWORD_QUALITY_NUMERIC_COMPLEX` Constant for `setPasswordQuality(ComponentName, int)` : the user must have entered a password containing at least numeric characters with no repeating (4444) or ordered (1234, 4321, 2468) sequences.
`int`	`PASSWORD_QUALITY_SOMETHING` Constant for `setPasswordQuality(ComponentName, int)` : the policy requires some kind of password or pattern, but doesn't care what it is.
`int`	`PASSWORD_QUALITY_UNSPECIFIED` Constant for `setPasswordQuality(ComponentName, int)` : the policy has no requirements for the password.

`int`	`PERMISSION_GRANT_STATE_DEFAULT` Runtime permission state: The user can manage the permission through the UI.
`int`	`PERMISSION_GRANT_STATE_DENIED` Runtime permission state: The permission is denied to the app and the user cannot manage the permission through the UI.
`int`	`PERMISSION_GRANT_STATE_GRANTED` Runtime permission state: The permission is granted to the app and the user cannot manage the permission through the UI.
`int`	`PERMISSION_POLICY_AUTO_DENY` Permission policy to always deny new permission requests for runtime permissions.
`int`	`PERMISSION_POLICY_AUTO_GRANT` Permission policy to always grant new permission requests for runtime permissions.
`int`	`PERMISSION_POLICY_PROMPT` Permission policy to prompt user for new permission requests for runtime permissions.
`int`	`PERSONAL_APPS_NOT_SUSPENDED` Return value for `getPersonalAppsSuspendedReasons(ComponentName)` when personal apps are not suspended.
`int`	`PERSONAL_APPS_SUSPENDED_EXPLICITLY` Flag for `getPersonalAppsSuspendedReasons(ComponentName)` return value.
`int`	`PERSONAL_APPS_SUSPENDED_PROFILE_TIMEOUT` Flag for `getPersonalAppsSuspendedReasons(ComponentName)` return value.
`String`	`POLICY_DISABLE_CAMERA` Constant to indicate the feature of disabling the camera.

`String`	`POLICY_DISABLE_SCREEN_CAPTURE` Constant to indicate the feature of disabling screen captures.
`int`	`PRIVATE_DNS_MODE_OFF` Specifies that Private DNS was turned off completely.
`int`	`PRIVATE_DNS_MODE_OPPORTUNISTIC` Specifies that the device owner requested opportunistic DNS over TLS
`int`	`PRIVATE_DNS_MODE_PROVIDER_HOSTNAME` Specifies that the device owner configured a specific host to use for Private DNS.
`int`	`PRIVATE_DNS_MODE_UNKNOWN` Specifies that the Private DNS setting is in an unknown state.
`int`	`PRIVATE_DNS_SET_ERROR_FAILURE_SETTING` General failure to set the Private DNS mode, not due to one of the reasons listed above.
`int`	`PRIVATE_DNS_SET_ERROR_HOST_NOT_SERVING` If the `privateDnsHost` provided was of a valid hostname but that host was found to not support DNS-over-TLS.
`int`	`PRIVATE_DNS_SET_NO_ERROR` The selected mode has been set successfully.
`int`	`PROVISIONING_MODE_FULLY_MANAGED_DEVICE` The provisioning mode for fully managed device.
`int`	`PROVISIONING_MODE_MANAGED_PROFILE` The provisioning mode for managed profile.
`int`	`PROVISIONING_MODE_MANAGED_PROFILE_ON_PERSONAL_DEVICE` The provisioning mode for a managed profile on a personal device.

int	`RESET_PASSWORD_DO_NOT_ASK_CREDENTIALS_ON_BOOT` Flag for `resetPasswordWithToken(ComponentName, String, byte, int)` and `resetPassword(String, int)` : don't ask for user credentials on device boot.
int	`RESET_PASSWORD_REQUIRE_ENTRY` Flag for `resetPasswordWithToken(ComponentName, String, byte, int)` and `resetPassword(String, int)` : don't allow other admins to change the password again until the user has entered it.
int	`SKIP_SETUP_WIZARD` Flag used by `createAndManageUser(ComponentName, String, ComponentName, PersistableBundle, int)` to skip setup wizard after creating a new user.
int	`WIFI_SECURITY_ENTERPRISE_192` Constant for `getMinimumRequiredWifiSecurityLevel()` and `setMinimumRequiredWifiSecurityLevel(int)` : enterprise 192 bit network.
int	`WIFI_SECURITY_ENTERPRISE_EAP` Constant for `getMinimumRequiredWifiSecurityLevel()` and `setMinimumRequiredWifiSecurityLevel(int)` : enterprise EAP network.
int	`WIFI_SECURITY_OPEN` Constant for `getMinimumRequiredWifiSecurityLevel()` and `setMinimumRequiredWifiSecurityLevel(int)` : no minimum security level.
int	`WIFI_SECURITY_PERSONAL` Constant for `getMinimumRequiredWifiSecurityLevel()` and `setMinimumRequiredWifiSecurityLevel(int)` : personal network such as WEP, WPA2-PSK.
int	`WIPE_EUICC` Flag for `wipeDevice(int)` : also erase the device's eUICC data.
int	`WIPE_EXTERNAL_STORAGE` Flag for `wipeData(int)` : also erase the device's adopted external storage (such as adopted SD cards).

`int`	`WIPE_RESET_PROTECTION_DATA` Flag for `wipeData(int)` : also erase the factory reset protection data.
`int`	`WIPE_SILENTLY` Flag for `wipeData(int)` : won't show reason for wiping to the user.

Public methods

`void`	`acknowledgeDeviceCompliant()` Called by a profile owner of an organization-owned managed profile to acknowledge that the device is compliant and the user can turn the profile off if needed according to the maximum time off policy.
`void`	`addCrossProfileIntentFilter(ComponentName admin, IntentFilter filter, int flags)` Called by the profile owner of a managed profile so that some intents sent in the managed profile can also be resolved in the parent, or vice versa.
`boolean`	`addCrossProfileWidgetProvider(ComponentName admin, String packageName)` Called by the profile owner of a managed profile or a holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_PROFILE_INTERACTION` to enable widget providers from a given package to be available in the parent profile.
`int`	`addOverrideApn(ComponentName admin, ApnSetting apnSetting)` Called by device owner or managed profile owner to add an override APN.
`void`	`addPersistentPreferredActivity(ComponentName admin, IntentFilter filter, ComponentName activity)` Called by a profile owner or device owner or holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_LOCK_TASK` .
`void`	`addUserRestriction(ComponentName admin, String key)` Called by a profile owner, device owner or a holder of any permission that is associated with a user restriction to set a user restriction specified by the key.
`void`	`addUserRestrictionGlobally(String key)`

	Called by a profile owner, device owner or a holder of any permission that is associated with a user restriction to set a user restriction specified by the provided `key` globally on all users.
`boolean`	`bindDeviceAdminServiceAsUser(ComponentName admin, Intent serviceIntent, ServiceConnection conn, int flags, UserHandle targetUser)` Called by a device owner to bind to a service from a secondary managed user or vice versa.
`boolean`	`bindDeviceAdminServiceAsUser(ComponentName admin, Intent serviceIntent, ServiceConnection conn, Context.BindServiceFlags flags, UserHandle targetUser)` See `bindDeviceAdminServiceAsUser(ComponentName,Intent,ServiceConnection,int,UserHandle)`.
`boolean`	`canAdminGrantSensorsPermissions()` Returns true if the caller is running on a device where an admin can grant permissions related to device sensors.
`boolean`	`canUsbDataSignalingBeDisabled()` Returns whether enabling or disabling USB data signaling is supported on the device.
`void`	`clearApplicationUserData(ComponentName admin, String packageName, Executor executor, DevicePolicyManager.OnClearApplicationUserDataListener listener)` Called by the device owner or profile owner to clear application user data of a given package.
`void`	`clearCrossProfileIntentFilters(ComponentName admin)` Called by a profile owner of a managed profile to remove the cross-profile intent filters that go from the managed profile to the parent, or from the parent to the managed profile.
`void`	`clearDeviceOwnerApp(String packageName)` *This method was deprecated in API level 26. This method is expected to be used for testing purposes only. The device owner will lose control of the device and its data after calling it. In order to protect any sensitive data that remains on the device, it is advised that the device owner factory resets the device instead of calling this method. See* `wipeData(int)`.

`void`	`clearPackagePersistentPreferredActivities(ComponentName admin, String packageName)` Called by a profile owner or device owner or holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_LOCK_TASK` to remove all persistent intent handler preferences associated with the given package that were set by `addPersistentPreferredActivity(ComponentName, IntentFilter, ComponentName)` .
`void`	`clearProfileOwner(ComponentName admin)` *This method was deprecated in API level 26. This method is expected to be used for testing purposes only. The profile owner will lose control of the user and its data after calling it. In order to protect any sensitive data that remains on this user, it is advised that the profile owner deletes it instead of calling this method. See* `wipeData(int)` .
`boolean`	`clearResetPasswordToken(ComponentName admin)` Called by a profile, device owner or holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_RESET_PASSWORD` to revoke the current password reset token.
`void`	`clearUserRestriction(ComponentName admin, String key)` Called by a profile owner, device owner or a holder of any permission that is associated with a user restriction to clear a user restriction specified by the key.
`Intent`	`createAdminSupportIntent(String restriction)` Called by any app to display a support dialog when a feature was disabled by an admin.
`UserHandle`	`createAndManageUser(ComponentName admin, String name, ComponentName profileOwner, PersistableBundle adminExtras, int flags)` Called by a device owner to create a user with the specified name and a given component of the calling package as profile owner.
`int`	`enableSystemApp(ComponentName admin, Intent intent)` Re-enable system apps by intent that were disabled by default when the user was initialized.
`void`	`enableSystemApp(ComponentName admin, String packageName)` Re-enable a system app that was disabled by default when the user was initialized.
`AttestedKeyPair`	`generateKeyPair(ComponentName admin, String algorithm, KeyGenParameterSpec keySpec, int idAttestationFlags)`

	This API can be called by the following to generate a new private/public key pair: • Device owner • Profile owner • Delegated certificate installer • Credential management app • An app that holds the `Manifest.permission.MANAGE_DEVICE_POLICY_CERTIFICATES` permission If the device supports key generation via secure hardware, this method is useful for creating a key in KeyChain that never left the secure hardware.
`String[]`	`getAccountTypesWithManagementDisabled()` Gets the array of accounts for which account management is disabled by the profile owner or device owner.
`List<ComponentName>`	`getActiveAdmins()` Return a list of all currently active device administrators' component names.
`Set<String>`	`getAffiliationIds(ComponentName admin)` Returns the set of affiliation ids previously set via `setAffiliationIds(ComponentName, Set)`, or an empty set if none have been set.
`Set<String>`	`getAlwaysOnVpnLockdownWhitelist(ComponentName admin)` Called by device or profile owner to query the set of packages that are allowed to access the network directly when always-on VPN is in lockdown mode but not connected.
`String`	`getAlwaysOnVpnPackage(ComponentName admin)` Called by a device or profile owner to read the name of the package administering an always-on VPN connection for the current user.
`int`	`getAppFunctionsPolicy()` Returns the current `AppFunctionManager` policy.
`Bundle`	`getApplicationRestrictions(ComponentName admin, String packageName)` Retrieves the application restrictions for a given target application running in the calling user.
`String`	`getApplicationRestrictionsManagingPackage(ComponentName admin)`

	This method was deprecated in API level 26. From `Build.VERSION_CODES.O` *. Use* `getDelegatePackages(ComponentName, String)` *with the* `DELEGATION_APP_RESTRICTIONS` *scope instead.*
`boolean`	`getAutoTimeEnabled(ComponentName admin)` Returns true if auto time is enabled on the device.
`int`	`getAutoTimePolicy()` Returns current auto time policy's state.
`boolean`	`getAutoTimeRequired()` *This method was deprecated in API level 30. From* `Build.VERSION_CODES.R` *. Use* `getAutoTimeEnabled(ComponentName)`
`boolean`	`getAutoTimeZoneEnabled(ComponentName admin)` Returns true if auto time zone is enabled on the device.
`int`	`getAutoTimeZonePolicy()` Returns auto time zone policy's current state.
`List<UserHandle>`	`getBindDeviceAdminTargetUsers(ComponentName admin)` Returns the list of target users that the calling device owner or owner of secondary user can use when calling `bindDeviceAdminServiceAsUser(ComponentName, Intent, ServiceConnection, BindServiceFlags, UserHandle)` .
`boolean`	`getBluetoothContactSharingDisabled(ComponentName admin)` Called by a profile owner of a managed profile to determine whether or not Bluetooth devices cannot access enterprise contacts.
`boolean`	`getCameraDisabled(ComponentName admin)` Determine whether or not the device's cameras have been disabled for this user, either by the calling admin, if specified, or all admins.

String	getCertInstallerPackage(ComponentName admin) *This method was deprecated in API level 26. From Build.VERSION_CODES.O . Use getDelegatePackages(ComponentName, String) with the DELEGATION_CERT_INSTALL scope instead.*
int	getContentProtectionPolicy(ComponentName admin) Returns the current content protection policy.
PackagePolicy	getCredentialManagerPolicy() Called by a device owner or profile owner of a managed profile to retrieve the credential manager policy.
Set<String>	getCrossProfileCalendarPackages(ComponentName admin) *This method was deprecated in API level 34. Use setCrossProfilePackages(ComponentName,Set) .*
boolean	getCrossProfileCallerIdDisabled(ComponentName admin) *This method was deprecated in API level 34. starting with Build.VERSION_CODES.UPSIDE_DOWN_CAKE , use getManagedProfileCallerIdAccessPolicy() instead*
boolean	getCrossProfileContactsSearchDisabled(ComponentName admin) *This method was deprecated in API level 34. From Build.VERSION_CODES.UPSIDE_DOWN_CAKE use getManagedProfileContactsAccessPolicy()*
Set<String>	getCrossProfilePackages(ComponentName admin) Returns the set of package names that the admin has previously set as allowed to request user consent for cross-profile communication, via setCrossProfilePackages(ComponentName,Set) .
List<String>	getCrossProfileWidgetProviders(ComponentName admin) Called by the profile owner of a managed profile or a holder of the permission Manifest.permission.MANAGE_DEVICE_POLICY_PROFILE_INTERACTION to query providers from which packages are available in the parent profile.

`int`	`getCurrentFailedPasswordAttempts()` Retrieve the number of times the user has failed at entering a password since that last successful password entry.
`List<String>`	`getDelegatePackages(ComponentName admin, String delegationScope)` Called by a profile owner or device owner to retrieve a list of delegate packages that were granted a delegation scope.
`List<String>`	`getDelegatedScopes(ComponentName admin, String delegatedPackage)` Called by a profile owner or device owner to retrieve a list of the scopes given to a delegate package.
`CharSequence`	`getDeviceOwnerLockScreenInfo()`
`String`	`getDevicePolicyManagementRoleHolderPackage()` Returns the package name of the device policy management role holder.
`CharSequence`	`getEndUserSessionMessage(ComponentName admin)` Returns the user session end message.
`String`	`getEnrollmentSpecificId()` Returns an enrollment-specific identifier of this device, which is guaranteed to be the same value for the same device, enrolled into the same organization by the same managing app.
`FactoryResetProtectionPolicy`	`getFactoryResetProtectionPolicy(ComponentName admin)` Callable by device owner or profile owner of an organization-owned device, to retrieve the current factory reset protection (FRP) policy set previously by `setFactoryResetProtectionPolicy(ComponentName, FactoryResetProtectionPolicy)` .
`String`	`getGlobalPrivateDnsHost(ComponentName admin)` Returns the system-wide Private DNS host.
`int`	`getGlobalPrivateDnsMode(ComponentName admin)` Returns the system-wide Private DNS mode.
`List<byte[]>`	`getInstalledCaCerts(ComponentName admin)`

	Returns all CA certificates that are currently trusted, excluding system CA certificates.
`List<String>`	`getKeepUninstalledPackages(ComponentName admin)` Get the list of apps to keep around as APKs even if no user has currently installed it.
`Map<Integer, Set<String>>`	`getKeyPairGrants(String alias)` Called by a device or profile owner, or delegated certificate chooser (an app that has been delegated the `DELEGATION_CERT_SELECTION` privilege), to query which apps have access to a given KeyChain key.
`int`	`getKeyguardDisabledFeatures(ComponentName admin)` Determine whether or not features have been disabled in keyguard either by the calling admin, if specified, or all admins that set restrictions on this user and its participating profiles.
`int`	`getLockTaskFeatures(ComponentName admin)` Gets which system features are enabled for LockTask mode.
`String[]`	`getLockTaskPackages(ComponentName admin)` Returns the list of packages allowed to start the lock task mode.
`CharSequence`	`getLongSupportMessage(ComponentName admin)` Called by a device admin to get the long support message.
`PackagePolicy`	`getManagedProfileCallerIdAccessPolicy()` Called by a profile owner of a managed profile to retrieve the caller id policy.
`PackagePolicy`	`getManagedProfileContactsAccessPolicy()` Called by a profile owner of a managed profile to determine the current policy applied to managed profile contacts.
`long`	`getManagedProfileMaximumTimeOff(ComponentName admin)` Called by a profile owner of an organization-owned managed profile to get maximum time the profile is allowed to be turned off.

`ManagedSubscriptionsPolicy`	`getManagedSubscriptionsPolicy()` Returns the current `ManagedSubscriptionsPolicy`.
`int`	`getMaximumFailedPasswordsForWipe(ComponentName admin)` Retrieve the current maximum number of login attempts that are allowed before the device or profile is wiped, for a particular admin or all admins that set restrictions on this user and its participating profiles.
`long`	`getMaximumTimeToLock(ComponentName admin)` Retrieve the current maximum time to unlock for a particular admin or all admins that set restrictions on this user and its participating profiles.
`List<String>`	`getMeteredDataDisabledPackages(ComponentName admin)` Called by a device or profile owner to retrieve the list of packages which are restricted by the admin from using metered data.
`int`	`getMinimumRequiredWifiSecurityLevel()` Returns the current Wi-Fi minimum security level.
`int`	`getMtePolicy()` Called by a device owner, profile owner of an organization-owned device to get the Memory Tagging Extension (MTE) policy Learn more about MTE
`int`	`getNearbyAppStreamingPolicy()` Returns the current runtime nearby app streaming policy set by the device or profile owner.
`int`	`getNearbyNotificationStreamingPolicy()` Returns the current runtime nearby notification streaming policy set by the device or profile owner.
`int`	`getOrganizationColor(ComponentName admin)` *This method was deprecated in API level 31. From `Build.VERSION_CODES.R`, the organization color is never used as the background color of the confirm credentials screen.*
`CharSequence`	`getOrganizationName(ComponentName admin)`

	Called by the device owner (since API 26) or profile owner (since API 24) or holders of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_ORGANIZATION_IDENTITY` to retrieve the name of the organization under management.
`List<ApnSetting>`	`getOverrideApns(ComponentName admin)` Called by device owner or managed profile owner to get all override APNs inserted by device owner or managed profile owner previously using `addOverrideApn(ComponentName, ApnSetting)` .
`DevicePolicyManager`	`getParentProfileInstance(ComponentName admin)` Called by the profile owner of a managed profile or other apps in a managed profile to obtain a `DevicePolicyManager` whose calls act on the parent profile.
`int`	`getPasswordComplexity()` Returns how complex the current user's screen lock is.
`long`	`getPasswordExpiration(ComponentName admin)` Get the current password expiration time for a particular admin or all admins that set restrictions on this user and its participating profiles.
`long`	`getPasswordExpirationTimeout(ComponentName admin)` Get the password expiration timeout for the given admin.
`int`	`getPasswordHistoryLength(ComponentName admin)` Retrieve the current password history length for a particular admin or all admins that set restrictions on this user and its participating profiles.
`int`	`getPasswordMaximumLength(int quality)` Return the maximum password length that the device supports for a particular password quality.
`int`	`getPasswordMinimumLength(ComponentName admin)` *This method was deprecated in API level 31. see `setPasswordQuality(ComponentName,int)` for details.*

`int`	`getPasswordMinimumLetters(ComponentName admin)` *This method was deprecated in API level 31. see* `setPasswordQuality(ComponentName,int)` *for details.*
`int`	`getPasswordMinimumLowerCase(ComponentName admin)` *This method was deprecated in API level 31. see* `setPasswordQuality(ComponentName,int)` *for details.*
`int`	`getPasswordMinimumNonLetter(ComponentName admin)` *This method was deprecated in API level 31. see* `setPasswordQuality(ComponentName,int)` *for details.*
`int`	`getPasswordMinimumNumeric(ComponentName admin)` *This method was deprecated in API level 31. see* `setPasswordQuality(ComponentName,int)` *for details.*
`int`	`getPasswordMinimumSymbols(ComponentName admin)` *This method was deprecated in API level 31. see* `setPasswordQuality(ComponentName,int)` *for details.*
`int`	`getPasswordMinimumUpperCase(ComponentName admin)` *This method was deprecated in API level 31. see* `setPasswordQuality(ComponentName,int)` *for details.*
`int`	`getPasswordQuality(ComponentName admin)` *This method was deprecated in API level 31. see* `setPasswordQuality(ComponentName,int)` *for details.*
`SystemUpdateInfo`	`getPendingSystemUpdate(ComponentName admin)` Get information about a pending system update.
`int`	`getPermissionGrantState(ComponentName admin, String packageName, String permission)` Returns the current grant state of a runtime permission for a specific application.

`int`	`getPermissionPolicy(ComponentName admin)` Returns the current runtime permission policy set by the device or profile owner.
`List<String>`	`getPermittedAccessibilityServices(ComponentName admin)` Returns the list of permitted accessibility services set by this device or profile owner.
`List<String>`	`getPermittedCrossProfileNotificationListeners(ComponentName admin)` Returns the list of packages installed on the primary user that allowed to use a `NotificationListenerService` to receive notifications from this managed profile, as set by the profile owner.
`List<String>`	`getPermittedInputMethods(ComponentName admin)` Returns the list of permitted input methods set by this device or profile owner.
`int`	`getPersonalAppsSuspendedReasons(ComponentName admin)` Called by profile owner of an organization-owned managed profile to check whether personal apps are suspended.
`List<PreferentialNetworkServiceConfig>`	`getPreferentialNetworkServiceConfigs()` Get preferential network configuration
`int`	`getRequiredPasswordComplexity()` Gets the password complexity requirement set by `setRequiredPasswordComplexity(int)`, for the current user.
`long`	`getRequiredStrongAuthTimeout(ComponentName admin)` Determine for how long the user will be able to use secondary, non strong auth for authentication, since last strong method authentication (password, pin or pattern) was used.
`DevicePolicyResourcesManager`	`getResources()` Returns a `DevicePolicyResourcesManager` containing the required APIs to set, reset, and get device policy related resources.

`boolean`	`getScreenCaptureDisabled(ComponentName admin)` Determine whether or not screen capture has been disabled by the calling admin, if specified, or all admins.
`List<UserHandle>`	`getSecondaryUsers(ComponentName admin)` Called by a device owner to list all secondary users on the device.
`CharSequence`	`getShortSupportMessage(ComponentName admin)` Called by a device admin or holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_SUPPORT_MESSAGE` to get the short support message.
`CharSequence`	`getStartUserSessionMessage(ComponentName admin)` Returns the user session start message.
`boolean`	`getStorageEncryption(ComponentName admin)` *This method was deprecated in API level 30. This method only returns the value set by `setStorageEncryption(ComponentName, boolean)`. It does not actually reflect the storage encryption status. Use `getStorageEncryptionStatus()` for that. Called by an application that is administering the device to determine the requested setting for secure storage.*
`int`	`getStorageEncryptionStatus()` Called by an application that is administering the device to determine the current encryption status of the device.
`Set<Integer>`	`getSubscriptionIds()` Returns the subscription ids of all subscriptions which were downloaded by the calling admin.
`SystemUpdatePolicy`	`getSystemUpdatePolicy()` Retrieve a local system update policy set previously by `setSystemUpdatePolicy(ComponentName, SystemUpdatePolicy)`.
`PersistableBundle`	`getTransferOwnershipBundle()` Returns the data passed from the current administrator to the new administrator during an ownership transfer.

List<PersistableBundle>	getTrustAgentConfiguration(ComponentName admin, ComponentName agent) Gets configuration for the given trust agent based on aggregating all calls to setTrustAgentConfiguration(ComponentName,ComponentName,PersistableBundle) for all device admins.
List<String>	getUserControlDisabledPackages(ComponentName admin) Returns the list of packages over which user control is disabled by a device or profile owner or holders of the permission Manifest.permission.MANAGE_DEVICE_POLICY_APPS_CONTROL.
Bundle	getUserRestrictions(ComponentName admin) Called by an admin to get user restrictions set by themselves with addUserRestriction(ComponentName,String).
Bundle	getUserRestrictionsGlobally() Called by a profile or device owner to get global user restrictions set with addUserRestrictionGlobally(String).
String	getWifiMacAddress(ComponentName admin) Called by a device owner or profile owner on organization-owned device to get the MAC address of the Wi-Fi device.
WifiSsidPolicy	getWifiSsidPolicy() Returns the current Wi-Fi SSID policy.
boolean	grantKeyPairToApp(ComponentName admin, String alias, String packageName) Called by a device or profile owner, or delegated certificate chooser (an app that has been delegated the DELEGATION_CERT_SELECTION privilege), to grant an application access to an already-installed (or generated) KeyChain key.
boolean	grantKeyPairToWifiAuth(String alias) Called by a device or profile owner, or delegated certificate chooser (an app that has been delegated the DELEGATION_CERT_SELECTION privilege), to allow using a KeyChain key pair for authentication to Wifi networks.
boolean	hasCaCertInstalled(ComponentName admin, byte[] certBuffer) Returns whether this certificate is installed as a trusted CA.

`boolean`	`hasGrantedPolicy(ComponentName admin, int usesPolicy)` Returns true if an administrator has been granted a particular device policy.
`boolean`	`hasKeyPair(String alias)` This API can be called by the following to query whether a certificate and private key are installed under a given alias: • Device owner • Profile owner • Delegated certificate installer • Credential management app • An app that holds the `Manifest.permission.MANAGE_DEVICE_POLICY_CERTIFICATES` permission If called by the credential management app, the alias must exist in the credential management app's `AppUriAuthenticationPolicy` .
`boolean`	`hasLockdownAdminConfiguredNetworks(ComponentName admin)` Called by a device owner or a profile owner of an organization-owned managed profile to determine whether the user is prevented from modifying networks configured by the admin.
`boolean`	`installCaCert(ComponentName admin, byte[] certBuffer)` Installs the given certificate as a user CA.
`boolean`	`installExistingPackage(ComponentName admin, String packageName)` Install an existing package that has been installed in another user, or has been kept after removal via `setKeepUninstalledPackages(ComponentName, List)` .
`boolean`	`installKeyPair(ComponentName admin, PrivateKey privKey, Certificate[] certs, String alias, int flags)` This API can be called by the following to install a certificate chain and corresponding private key for the leaf certificate: • Device owner • Profile owner • Delegated certificate installer • Credential management app • An app that holds the `Manifest.permission.MANAGE_DEVICE_POLICY_CERTIFICATES` permission

	All apps within the profile will be able to access the certificate chain and use the private key, given direct user approval (if the user is allowed to select the private key).
`boolean`	`installKeyPair(ComponentName admin, PrivateKey privKey, Certificate[] certs, String alias, boolean requestAccess)` This API can be called by the following to install a certificate chain and corresponding private key for the leaf certificate: • Device owner • Profile owner • Delegated certificate installer • Credential management app • An app that holds the `Manifest.permission.MANAGE_DEVICE_POLICY_CERTIFICATES` permission All apps within the profile will be able to access the certificate chain and use the private key, given direct user approval.
`boolean`	`installKeyPair(ComponentName admin, PrivateKey privKey, Certificate cert, String alias)` This API can be called by the following to install a certificate and corresponding private key: • Device owner • Profile owner • Delegated certificate installer • Credential management app • An app that holds the `Manifest.permission.MANAGE_DEVICE_POLICY_CERTIFICATES` permission All apps within the profile will be able to access the certificate and use the private key, given direct user approval.
`void`	`installSystemUpdate(ComponentName admin, Uri updateFilePath, Executor executor, DevicePolicyManager.InstallSystemUpdateCallback callback)` Called by device owner or profile owner of an organization-owned managed profile to install a system update from the given file.
`boolean`	`isActivePasswordSufficient()` Determines whether the calling user's current password meets policy requirements (e.g. quality, minimum length).
`boolean`	`isActivePasswordSufficientForDeviceRequirement()`

	Called by profile owner of a managed profile to determine whether the current device password meets policy requirements set explicitly device-wide.
`boolean`	`isAdminActive(ComponentName admin)` Return true if the given administrator component is currently active (enabled) in the system.
`boolean`	`isAffiliatedUser()` Returns whether this user is affiliated with the device.
`boolean`	`isAlwaysOnVpnLockdownEnabled(ComponentName admin)` Called by device or profile owner to query whether current always-on VPN is configured in lockdown mode.
`boolean`	`isApplicationHidden(ComponentName admin, String packageName)` Determine if a package is hidden.
`boolean`	`isBackupServiceEnabled(ComponentName admin)` Return whether the backup service is enabled by the device owner or profile owner for the current user, as previously set by `setBackupServiceEnabled(ComponentName,boolean)` .
`boolean`	`isCallerApplicationRestrictionsManagingPackage()` *This method was deprecated in API level 26. From* `Build.VERSION_CODES.O` *. Use* `getDelegatedScopes(ComponentName, String)` *instead.*
`boolean`	`isCommonCriteriaModeEnabled(ComponentName admin)` Returns whether Common Criteria mode is currently enabled.
`boolean`	`isComplianceAcknowledgementRequired()` Called by a profile owner of an organization-owned managed profile to query whether it needs to acknowledge device compliance to allow the user to turn the profile off if needed according to the maximum profile time off policy.
`boolean`	`isDeviceFinanced()` Returns `true` if this device is marked as a financed device.
`boolean`	`isDeviceIdAttestationSupported()`

	Returns `true` if the device supports attestation of device identifiers in addition to key attestation.
`boolean`	`isDeviceOwnerApp(String packageName)` Used to determine if a particular package has been registered as a Device Owner app.
`boolean`	`isEphemeralUser(ComponentName admin)` Checks if the profile owner is running in an ephemeral user.
`boolean`	`isKeyPairGrantedToWifiAuth(String alias)` Called by a device or profile owner, or delegated certificate chooser (an app that has been delegated the `DELEGATION_CERT_SELECTION` privilege), to query whether a KeyChain key pair can be used for authentication to Wifi networks.
`boolean`	`isLockTaskPermitted(String pkg)` This function lets the caller know whether the given component is allowed to start the lock task mode.
`boolean`	`isLogoutEnabled()` Returns whether logout is enabled by a device owner.
`boolean`	`isManagedProfile(ComponentName admin)` Return if this user is a managed profile of another user.
`boolean`	`isMasterVolumeMuted(ComponentName admin)` Called by profile or device owners to check whether the global volume mute is on or off.
`static boolean`	`isMtePolicyEnforced()` Get the current MTE state of the device.
`boolean`	`isNetworkLoggingEnabled(ComponentName admin)` Return whether network logging is enabled by a device owner or profile owner of a managed profile.
`boolean`	`isOrganizationOwnedDeviceWithManagedProfile()`

	Apps can use this method to find out if the device was provisioned as organization-owend device with a managed profile.
`boolean`	`isOverrideApnEnabled(ComponentName admin)` Called by device owner to check if override APNs are currently enabled.
`boolean`	`isPackageSuspended(ComponentName admin, String packageName)` Determine if a package is suspended.
`boolean`	`isPreferentialNetworkServiceEnabled()` Indicates whether preferential network service is enabled.
`boolean`	`isProfileOwnerApp(String packageName)` Used to determine if a particular package is registered as the profile owner for the user.
`boolean`	`isProvisioningAllowed(String action)` Returns whether it is possible for the caller to initiate provisioning of a managed profile or device, setting itself as the device or profile owner.
`boolean`	`isResetPasswordTokenActive(ComponentName admin)` Called by a profile, device owner or a holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_RESET_PASSWORD` to check if the current reset password token is active.
`boolean`	`isSafeOperation(int reason)` Checks if it's safe to run operations that can be affected by the given `reason`.
`boolean`	`isSecurityLoggingEnabled(ComponentName admin)` Return whether security logging is enabled or not by the admin.
`boolean`	`isStatusBarDisabled()` Returns whether the status bar is disabled/enabled, see `setStatusBarDisabled(ComponentName, boolean)`.

`boolean`	`isUninstallBlocked(ComponentName admin, String packageName)` Check whether the user has been blocked by device policy from uninstalling a package.
`boolean`	`isUniqueDeviceAttestationSupported()` Returns `true` if the StrongBox Keymaster implementation on the device was provisioned with an individual attestation certificate and can sign attestation records using it (as attestation using an individual attestation certificate is a feature only Keymaster implementations with StrongBox security level can implement).
`boolean`	`isUsbDataSignalingEnabled()` Returns whether USB data signaling is currently enabled.
`boolean`	`isUsingUnifiedPassword(ComponentName admin)` When called by a profile owner of a managed profile returns true if the profile uses unified challenge with its parent user.
`List<UserHandle>`	`listForegroundAffiliatedUsers()` Gets the list of `affiliated` users running on foreground.
`void`	`lockNow()` Make the device lock immediately, as if the lock screen timeout has expired at the point of this call.
`void`	`lockNow(int flags)` Make the device lock immediately, as if the lock screen timeout has expired at the point of this call.
`int`	`logoutUser(ComponentName admin)` Called by a profile owner of secondary user that is affiliated with the device to stop the calling user and switch back to primary user (when the user was `switchUser(ComponentName,UserHandle)` switched to) or stop the user (when it was `started in background`).
`void`	`reboot(ComponentName admin)` Called by device owner to reboot the device.

`void`	`removeActiveAdmin(ComponentName admin)` Remove a current administration component.
`boolean`	`removeCrossProfileWidgetProvider(ComponentName admin, String packageName)` *This method was deprecated in API level 37. While this API still works to mutate the current allowlist, please consider switching to `setCrossProfileWidgetProviders(Set)` for better performance.*
`boolean`	`removeKeyPair(ComponentName admin, String alias)` This API can be called by the following to remove a certificate and private key pair installed under a given alias: • Device owner • Profile owner • Delegated certificate installer • Credential management app From Android `Build.VERSION_CODES.S`, the credential management app can call this API.
`boolean`	`removeOverrideApn(ComponentName admin, int apnId)` Called by device owner or managed profile owner to remove an override APN.
`boolean`	`removeUser(ComponentName admin, UserHandle userHandle)` Called by a device owner to remove a user/profile and all associated data.
`boolean`	`requestBugreport(ComponentName admin)` Called by a device owner to request a bugreport.
`boolean`	`resetPassword(String password, int flags)` *This method was deprecated in API level 30. Please use `resetPasswordWithToken(ComponentName, String, byte, int)` instead.*
`boolean`	`resetPasswordWithToken(ComponentName admin, String password, byte[] token, int flags)` Called by device or profile owner to force set a new device unlock password or a managed profile challenge on current user.

`List<NetworkEvent>`	`retrieveNetworkLogs(ComponentName admin, long batchToken)` Called by device owner, profile owner of a managed profile or delegated app with `DELEGATION_NETWORK_LOGGING` to retrieve the most recent batch of network logging events.
`List<SecurityLog.SecurityEvent>`	`retrievePreRebootSecurityLogs(ComponentName admin)` Called by device owner or profile owner of an organization-owned managed profile to retrieve device logs from before the device's last reboot.
`List<SecurityLog.SecurityEvent>`	`retrieveSecurityLogs(ComponentName admin)` Called by device owner or profile owner of an organization-owned managed profile to retrieve all new security logging entries since the last call to this API after device boots.
`boolean`	`revokeKeyPairFromApp(ComponentName admin, String alias, String packageName)` Called by a device or profile owner, or delegated certificate chooser (an app that has been delegated the `DELEGATION_CERT_SELECTION` privilege), to revoke an application's grant to a KeyChain key pair.
`boolean`	`revokeKeyPairFromWifiAuth(String alias)` Called by a device or profile owner, or delegated certificate chooser (an app that has been delegated the `DELEGATION_CERT_SELECTION` privilege), to deny using a KeyChain key pair for authentication to Wifi networks.
`void`	`setAccountManagementDisabled(ComponentName admin, String accountType, boolean disabled)` Called by a device owner or profile owner to disable account management for a specific type of account.
`void`	`setAffiliationIds(ComponentName admin, Set<String> ids)` Indicates the entity that controls the device.
`void`	`setAlwaysOnVpnPackage(ComponentName admin, String vpnPackage, boolean lockdownEnabled)` Called by a device or profile owner to configure an always-on VPN connection through a specific application for the current user.
`void`	`setAlwaysOnVpnPackage(ComponentName admin, String vpnPackage, boolean lockdownEnabled, Set<String> lockdownAllowlist)`

	A version of `setAlwaysOnVpnPackage(ComponentName,String,boolean)` that allows the admin to specify a set of apps that should be able to access the network directly when VPN is not connected.
`void`	`setAppFunctionsPolicy(int policy)` Sets the `AppFunctionManager` policy which controls app functions operations on the device.
`boolean`	`setApplicationHidden(ComponentName admin, String packageName, boolean hidden)` Hide or unhide packages.
`void`	`setApplicationRestrictions(ComponentName admin, String packageName, Bundle settings)` Sets the application restrictions for a given target application running in the calling user.
`void`	`setApplicationRestrictionsManagingPackage(ComponentName admin, String packageName)` *This method was deprecated in API level 26. From* `Build.VERSION_CODES.O` *. Use* `setDelegatedScopes(ComponentName, String, List)` *with the* `DELEGATION_APP_RESTRICTIONS` *scope instead.*
`void`	`setAutoTimeEnabled(ComponentName admin, boolean enabled)` Called by a device owner, a profile owner for the primary user or a profile owner of an organization-owned managed profile to turn auto time on and off.
`void`	`setAutoTimePolicy(int policy)` Called by a device owner, a profile owner for the primary user or a profile owner of an organization-owned managed profile to turn auto time on and off i.e.
`void`	`setAutoTimeRequired(ComponentName admin, boolean required)` *This method was deprecated in API level 30. From* `Build.VERSION_CODES.R` *. Use* `setAutoTimeEnabled(ComponentName, boolean)` *to turn auto time on or off and use* `UserManager.DISALLOW_CONFIG_DATE_TIME` *to prevent the user from changing this setting.*
`void`	`setAutoTimeZoneEnabled(ComponentName admin, boolean enabled)` Called by a device owner, a profile owner for the primary user or a profile owner of an organization-owned managed profile to turn auto time zone on and off.
`void`	`setAutoTimeZonePolicy(int policy)`

	Called by a device owner, a profile owner for the primary user or a profile owner of an organization-owned managed profile to turn auto time zone on and off.
`void`	`setBackupServiceEnabled(ComponentName admin, boolean enabled)` Allows the device owner or profile owner to enable or disable the backup service.
`void`	`setBluetoothContactSharingDisabled(ComponentName admin, boolean disabled)` Called by a profile owner of a managed profile to set whether bluetooth devices can access enterprise contacts.
`void`	`setCameraDisabled(ComponentName admin, boolean disabled)` Called by an application that is administering the device to disable all cameras on the device, for this user.
`void`	`setCertInstallerPackage(ComponentName admin, String installerPackage)` *This method was deprecated in API level 26. From* `Build.VERSION_CODES.O` *. Use* `setDelegatedScopes(ComponentName, String, List)` *with the* `DELEGATION_CERT_INSTALL` *scope instead.*
`void`	`setCommonCriteriaModeEnabled(ComponentName admin, boolean enabled)` Called by device owner or profile owner of an organization-owned managed profile to toggle Common Criteria mode for the device.
`void`	`setConfiguredNetworksLockdownState(ComponentName admin, boolean lockdown)` Called by a device owner or a profile owner of an organization-owned managed profile to control whether the user can change networks configured by the admin.
`void`	`setContentProtectionPolicy(ComponentName admin, int policy)` Sets the content protection policy which controls scanning for deceptive apps.
`void`	`setCredentialManagerPolicy(PackagePolicy policy)` Called by a device owner or profile owner of a managed profile to set the credential manager policy.

`void`	`setCrossProfileCalendarPackages(ComponentName admin, Set<String> packageNames)` *This method was deprecated in API level 34. Use* `setCrossProfilePackages(ComponentName,Set)` .
`void`	`setCrossProfileCallerIdDisabled(ComponentName admin, boolean disabled)` *This method was deprecated in API level 34. starting with* `Build.VERSION_CODES.UPSIDE_DOWN_CAKE` *, use* `setManagedProfileCallerIdAccessPolicy(PackagePolicy)` *instead*
`void`	`setCrossProfileContactsSearchDisabled(ComponentName admin, boolean disabled)` *This method was deprecated in API level 34. From* `Build.VERSION_CODES.UPSIDE_DOWN_CAKE` *use* `setManagedProfileContactsAccessPolicy(PackagePolicy)`
`void`	`setCrossProfilePackages(ComponentName admin, Set<String> packageNames)` Sets the set of admin-allowlisted package names that are allowed to request user consent for cross-profile communication.
`void`	`setCrossProfileWidgetProviders(Set<String> packageNames)` Called by the profile owner of a managed profile or a holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_PROFILE_INTERACTION` to enable widget providers from packages to be available in the parent profile.
`void`	`setDefaultDialerApplication(String packageName)` Must be called by a device owner or a profile owner of an organization-owned managed profile to set the default dialer application for the calling user.
`void`	`setDefaultSmsApplication(ComponentName admin, String packageName)` Must be called by a device owner or a profile owner of an organization-owned managed profile to set the default SMS application.
`void`	`setDelegatedScopes(ComponentName admin, String delegatePackage, List<String> scopes)` Called by a profile owner or device owner to grant access to privileged APIs to another app.
`void`	`setDeviceOwnerLockScreenInfo(ComponentName admin, CharSequence info)` Sets the device owner information to be shown on the lock screen.

`void`	`setEndUserSessionMessage(ComponentName admin, CharSequence endUserSessionMessage)` Called by a device owner to specify the user session end message.
`void`	`setFactoryResetProtectionPolicy(ComponentName admin, FactoryResetProtectionPolicy policy)` Callable by device owner or profile owner of an organization-owned device, to set a factory reset protection (FRP) policy.
`int`	`setGlobalPrivateDnsModeOpportunistic(ComponentName admin)` Sets the global Private DNS mode to opportunistic.
`int`	`setGlobalPrivateDnsModeSpecifiedHost(ComponentName admin, String privateDnsHost)` Sets the global Private DNS host to be used.
`void`	`setGlobalSetting(ComponentName admin, String setting, String value)` This method is mostly deprecated.
`void`	`setKeepUninstalledPackages(ComponentName admin, List<String> packageNames)` Set a list of apps to keep around as APKs even if no user has currently installed it.
`boolean`	`setKeyPairCertificate(ComponentName admin, String alias, List<Certificate> certs, boolean isUserSelectable)` This API can be called by the following to associate certificates with a key pair that was generated using `generateKeyPair(ComponentName, String, KeyGenParameterSpec, int)`, and set whether the key is available for the user to choose in the certificate selection prompt: • Device owner • Profile owner • Delegated certificate installer • Credential management app From Android `Build.VERSION_CODES.S`, the credential management app can call this API.
`boolean`	`setKeyguardDisabled(ComponentName admin, boolean disabled)` Called by a device owner or profile owner of secondary users that is affiliated with the device to disable the keyguard altogether.
`void`	`setKeyguardDisabledFeatures(ComponentName admin, int which)`

	Called by an application that is administering the device to disable keyguard customizations, such as widgets.
`void`	`setLocationEnabled(ComponentName admin, boolean locationEnabled)` Called by device owners to set the user's global location setting.
`void`	`setLockTaskFeatures(ComponentName admin, int flags)` Sets which system features are enabled when the device runs in lock task mode.
`void`	`setLockTaskPackages(ComponentName admin, String[] packages)` Sets which packages may enter lock task mode.
`void`	`setLogoutEnabled(ComponentName admin, boolean enabled)` Called by a device owner to specify whether logout is enabled for all secondary users.
`void`	`setLongSupportMessage(ComponentName admin, CharSequence message)` Called by a device admin to set the long support message.
`void`	`setManagedProfileCallerIdAccessPolicy(PackagePolicy policy)` Called by a profile owner of a managed profile to set the packages that are allowed to lookup contacts in the managed profile based on caller id information.
`void`	`setManagedProfileContactsAccessPolicy(PackagePolicy policy)` Called by a profile owner of a managed profile to set the packages that are allowed access to the managed profile contacts from the parent user.
`void`	`setManagedProfileMaximumTimeOff(ComponentName admin, long timeoutMillis)` Called by a profile owner of an organization-owned managed profile to set maximum time the profile is allowed to be turned off.
`void`	`setManagedSubscriptionsPolicy(ManagedSubscriptionsPolicy policy)` Called by a profile owner of an organization-owned device to specify `ManagedSubscriptionsPolicy` Managed subscriptions policy controls how SIMs would be associated with the managed profile.

`void`	`setMasterVolumeMuted(ComponentName admin, boolean on)` Called by profile or device owners to set the global volume mute on or off.
`void`	`setMaximumFailedPasswordsForWipe(ComponentName admin, int num)` Setting this to a value greater than zero enables a policy that will perform a device or profile wipe after too many incorrect device-unlock passwords have been entered.
`void`	`setMaximumTimeToLock(ComponentName admin, long timeMs)` Called by an application that is administering the device to set the maximum time for user activity until the device will lock.
`List<String>`	`setMeteredDataDisabledPackages(ComponentName admin, List<String> packageNames)` Called by a device or profile owner to restrict packages from using metered data.
`void`	`setMinimumRequiredWifiSecurityLevel(int level)` Called by device owner or profile owner of an organization-owned managed profile to specify the minimum security level required for Wi-Fi networks.
`void`	`setMtePolicy(int policy)` Called by a device owner, profile owner of an organization-owned device, to set the Memory Tagging Extension (MTE) policy.
`void`	`setNearbyAppStreamingPolicy(int policy)` Called by a device/profile owner to set nearby app streaming policy.
`void`	`setNearbyNotificationStreamingPolicy(int policy)` Called by a device/profile owner to set nearby notification streaming policy.
`void`	`setNetworkLoggingEnabled(ComponentName admin, boolean enabled)` Called by a device owner, profile owner of a managed profile or delegated app with `DELEGATION_NETWORK_LOGGING` to control the network logging feature.
`void`	`setOrganizationColor(ComponentName admin, int color)` *This method was deprecated in API level 31. From `Build.VERSION_CODES.R`, the organization color is never used as the background color of the confirm credentials screen.*

`void`	`setOrganizationId(String enterpriseId)` Sets the Enterprise ID for the work profile or managed device.
`void`	`setOrganizationName(ComponentName admin, CharSequence title)` Called by the device owner (since API 26) or profile owner (since API 24) to set the name of the organization under management.
`void`	`setOverrideApnsEnabled(ComponentName admin, boolean enabled)` Called by device owner to set if override APNs should be enabled.
`String[]`	`setPackagesSuspended(ComponentName admin, String[] packageNames, boolean suspended)` Called by device or profile owners to suspend packages for this user.
`void`	`setPasswordExpirationTimeout(ComponentName admin, long timeout)` Called by a device admin to set the password expiration timeout.
`void`	`setPasswordHistoryLength(ComponentName admin, int length)` Called by an application that is administering the device to set the length of the password history.
`void`	`setPasswordMinimumLength(ComponentName admin, int length)` *This method was deprecated in API level 31. see `setPasswordQuality(ComponentName,int)` for details.*
`void`	`setPasswordMinimumLetters(ComponentName admin, int length)` *This method was deprecated in API level 31. see `setPasswordQuality(ComponentName,int)` for details.*
`void`	`setPasswordMinimumLowerCase(ComponentName admin, int length)` *This method was deprecated in API level 31. see `setPasswordQuality(ComponentName,int)` for details.*
`void`	`setPasswordMinimumNonLetter(ComponentName admin, int length)` *This method was deprecated in API level 31. see `setPasswordQuality(ComponentName,int)` for details.*

`void`	`setPasswordMinimumNumeric(ComponentName admin, int length)` *This method was deprecated in API level 31. see `setPasswordQuality(ComponentName,int)` for details.*
`void`	`setPasswordMinimumSymbols(ComponentName admin, int length)` *This method was deprecated in API level 31. see `setPasswordQuality(ComponentName,int)` for details.*
`void`	`setPasswordMinimumUpperCase(ComponentName admin, int length)` *This method was deprecated in API level 31. see `setPasswordQuality(ComponentName,int)` for details.*
`void`	`setPasswordQuality(ComponentName admin, int quality)` *This method was deprecated in API level 31. Prefer using `setRequiredPasswordComplexity(int)`, to require a password that satisfies a complexity level defined by the platform, rather than specifying custom password requirement. Setting custom, overly-complicated password requirements leads to passwords that are hard for users to remember and may not provide any security benefits given as Android uses hardware-backed throttling to thwart online and offline brute-forcing of the device's screen lock. Company-owned devices (fully-managed and organization-owned managed profile devices) are able to continue using this method, though it is recommended that `setRequiredPasswordComplexity(int)` should be used instead.*
`boolean`	`setPermissionGrantState(ComponentName admin, String packageName, String permission, int grantState)` Sets the grant state of a runtime permission for a specific application.
`void`	`setPermissionPolicy(ComponentName admin, int policy)` Set the default response for future runtime permission requests by applications.
`boolean`	`setPermittedAccessibilityServices(ComponentName admin, List<String> packageNames)` Called by a profile or device owner to set the permitted `AccessibilityService`.
`boolean`	`setPermittedCrossProfileNotificationListeners(ComponentName admin, List<String> packageList)` Called by a profile owner of a managed profile to set the packages that are allowed to use a `NotificationListenerService` in the primary user to see notifications from the managed

	profile.
`boolean`	`setPermittedInputMethods(ComponentName admin, List<String> packageNames)` Called by a profile or device owner or holder of the `Manifest.permission.MANAGE_DEVICE_POLICY_INPUT_METHODS` permission to set the permitted input methods services for this user.
`void`	`setPersonalAppsSuspended(ComponentName admin, boolean suspended)` Called by a profile owner of an organization-owned managed profile to suspend personal apps on the device.
`void`	`setPreferentialNetworkServiceConfigs(List<PreferentialNetworkServiceConfig> preferentialNetworkServiceConfigs)` Sets preferential network configurations.
`void`	`setPreferentialNetworkServiceEnabled(boolean enabled)` Sets whether preferential network service is enabled.
`void`	`setProfileEnabled(ComponentName admin)` Sets the enabled state of the profile.
`void`	`setProfileName(ComponentName admin, String profileName)` Sets the name of the profile.
`void`	`setRecommendedGlobalProxy(ComponentName admin, ProxyInfo proxyInfo)` Set a network-independent global HTTP proxy.
`void`	`setRequiredPasswordComplexity(int passwordComplexity)` Sets a minimum password complexity requirement for the user's screen lock.
`void`	`setRequiredStrongAuthTimeout(ComponentName admin, long timeoutMs)` Called by a device/profile owner to set the timeout after which unlocking with secondary, non strong auth (e.g. fingerprint, face, trust agents) times out, i.e.
`boolean`	`setResetPasswordToken(ComponentName admin, byte[] token)`

	Called by a profile or device owner to provision a token which can later be used to reset the device lockscreen password (if called by device owner), or managed profile challenge (if called by profile owner), via `resetPasswordWithToken(ComponentName, String, byte, int)`.
`void`	`setRestrictionsProvider(ComponentName admin, ComponentName provider)` Designates a specific service component as the provider for making permission requests of a local or remote administrator of the user.
`void`	`setScreenCaptureDisabled(ComponentName admin, boolean disabled)` Called by a device/profile owner to set whether the screen capture is disabled.
`void`	`setSecureSetting(ComponentName admin, String setting, String value)` This method is mostly deprecated.
`void`	`setSecurityLoggingEnabled(ComponentName admin, boolean enabled)` Called by device owner or a profile owner of an organization-owned managed profile to control the security logging feature.
`void`	`setShortSupportMessage(ComponentName admin, CharSequence message)` Called by a device admin to set the short support message.
`void`	`setStartUserSessionMessage(ComponentName admin, CharSequence startUserSessionMessage)` Called by a device owner to specify the user session start message.
`boolean`	`setStatusBarDisabled(ComponentName admin, boolean disabled)` Called by device owner or profile owner of secondary users that is affiliated with the device to disable the status bar.
`int`	`setStorageEncryption(ComponentName admin, boolean encrypt)` *This method was deprecated in API level 30. This method does not actually modify the storage encryption of the device. It has never affected the encryption status of a device. Called by an application that is administering the device to request that the storage system be encrypted. Does nothing if the caller is on a secondary user or a managed profile.* *When multiple device administrators attempt to control device encryption, the most secure, supported setting will always be used. If any device administrator requests device encryption, it will be enabled; Conversely, if a device administrator attempts to disable device encryption*

	while another device administrator has enabled it, the call to disable will fail (most commonly returning `ENCRYPTION_STATUS_ACTIVE` *).* *This policy controls encryption of the secure (application data) storage area. Data written to other storage areas may or may not be encrypted, and this policy does not require or control the encryption of any other storage areas. There is one exception: If* `Environment.isExternalStorageEmulated()` *is* `true` *, then the directory returned by* `Environment.getExternalStorageDirectory()` *must be written to disk within the encrypted storage area.* *Important Note: On some devices, it is possible to encrypt storage without requiring the user to create a device PIN or Password. In this case, the storage is encrypted, but the encryption key may not be fully secured. For maximum security, the administrator should also require (and check for) a pattern, PIN, or password.*
`void`	`setSystemSetting(ComponentName admin, String setting, String value)` Called by a device or profile owner to update `Settings.System` settings.
`void`	`setSystemUpdatePolicy(ComponentName admin, SystemUpdatePolicy policy)` Called by device owners or profile owners of an organization-owned managed profile to set a local system update policy.
`boolean`	`setTime(ComponentName admin, long millis)` Called by a device owner or a profile owner of an organization-owned managed profile to set the system wall clock time.
`boolean`	`setTimeZone(ComponentName admin, String timeZone)` Called by a device owner or a profile owner of an organization-owned managed profile to set the system's persistent default time zone.
`void`	`setTrustAgentConfiguration(ComponentName admin, ComponentName target, PersistableBundle configuration)` Sets a list of configuration features to enable for a trust agent component.
`void`	`setUninstallBlocked(ComponentName admin, String packageName, boolean uninstallBlocked)` Change whether a user can uninstall a package.
`void`	`setUsbDataSignalingEnabled(boolean enabled)`

	Called by a device owner or profile owner of an organization-owned managed profile to enable or disable USB data signaling for the device.
`void`	`setUserControlDisabledPackages(ComponentName admin, List<String> packages)` Called by a device owner or a profile owner or holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_APPS_CONTROL` to disable user control over apps.
`void`	`setUserIcon(ComponentName admin, Bitmap icon)` Called by profile or device owners to set the user's photo.
`void`	`setWifiSsidPolicy(WifiSsidPolicy policy)` Called by device owner or profile owner of an organization-owned managed profile to specify the Wi-Fi SSID policy (`WifiSsidPolicy`).
`int`	`startUserInBackground(ComponentName admin, UserHandle userHandle)` Called by a device owner to start the specified secondary user in background.
`int`	`stopUser(ComponentName admin, UserHandle userHandle)` Called by a device owner to stop the specified secondary user.
`boolean`	`switchUser(ComponentName admin, UserHandle userHandle)` Called by a device owner to switch the specified secondary user to the foreground.
`void`	`transferOwnership(ComponentName admin, ComponentName target, PersistableBundle bundle)` Changes the current administrator to another one.
`void`	`uninstallAllUserCaCerts(ComponentName admin)` Uninstalls all custom trusted CA certificates from the profile.
`void`	`uninstallCaCert(ComponentName admin, byte[] certBuffer)` Uninstalls the given certificate from trusted user CAs, if present.
`boolean`	`updateOverrideApn(ComponentName admin, int apnId, ApnSetting apnSetting)` Called by device owner or managed profile owner to update an override APN.

`void`	`wipeData(int flags, CharSequence reason)` Ask that all user data be wiped.
`void`	`wipeData(int flags)` See `wipeData(int,CharSequence)`
`void`	`wipeDevice(int flags)` Ask that the device be wiped and factory reset.

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_ADD_DEVICE_ADMIN

```
public static final String ACTION_ADD_DEVICE_ADMIN
```

Activity action: ask the user to add a new device administrator to the system. The desired policy is the ComponentName of the policy in the `EXTRA_DEVICE_ADMIN` extra field. This will invoke a UI to bring the user through adding the device administrator to the system (or allowing them to reject it).

You can optionally include the `EXTRA_ADD_EXPLANATION` field to provide the user with additional explanation (in addition to your component's description) about what is being added.

If your administrator is already active, this will ordinarily return immediately (without user intervention). However, if your administrator has been updated and is requesting additional uses-policy flags, the user will be presented with the new list. New policies will not be available to the updated administrator until the user has accepted the new list.

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

ACTION_ADMIN_POLICY_COMPLIANCE

```
public static final String ACTION_ADMIN_POLICY_COMPLIANCE
```

Activity action: Starts the administrator to show policy compliance for the provisioning. This action is used any time that the administrator has an opportunity to show policy compliance before the end of setup wizard. This could happen

as part of the admin-integrated provisioning flow (in which case this gets sent after `ACTION_GET_PROVISIONING_MODE`), or it could happen during provisioning finalization if the administrator supports finalization during setup wizard.

Intents with this action may also be supplied with the `EXTRA_PROVISIONING_ADMIN_EXTRAS_BUNDLE` extra.

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

ACTION_APPLICATION_DELEGATION_SCOPES_CHANGED

```
public static final String ACTION_APPLICATION_DELEGATION_SCOPES_CHANGED
```

Broadcast Action: Sent after application delegation scopes are changed. The new delegation scopes will be sent in an `ArrayList<String>` extra identified by the `EXTRA_DELEGATION_SCOPES` key.

Note: This is a protected intent that can only be sent by the system.

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

ACTION_CHECK_POLICY_COMPLIANCE

```
public static final String ACTION_CHECK_POLICY_COMPLIANCE
```

Activity action: launch the DPC to check policy compliance. This intent is launched when the user taps on the notification about personal apps suspension. When handling this intent the DPC must check if personal apps should still be suspended and either unsuspend them or instruct the user on how to resolve the noncompliance causing the suspension.

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

ACTION_DEVICE_ADMIN_SERVICE

```
public static final String ACTION_DEVICE_ADMIN_SERVICE
```

Service action: Action for a service that device owner and profile owner can optionally own. If a device owner or a profile owner has such a service, the system tries to keep a bound connection to it, in order to keep their process always running. The service must be protected with the `Manifest.permission.BIND_DEVICE_ADMIN` permission.

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

ACTION_DEVICE_FINANCING_STATE_CHANGED

```
public static final String ACTION_DEVICE_FINANCING_STATE_CHANGED
```

Broadcast Action: Broadcast sent to indicate that the device financing state has changed.

This occurs when, for example, a financing kiosk app has been added or removed.

To query the current device financing state see `isDeviceFinanced()`.

This will be delivered to the following apps if they include a receiver for this action in their manifest:

- Device owner admins.
- Organization-owned profile owner admins
- The supervision app
- The device management role holder

Constant Value: "android.app.admin.action.DEVICE_FINANCING_STATE_CHANGED"

ACTION_MANAGED_PROFILE_PROVISIONED

```
public static final String ACTION_MANAGED_PROFILE_PROVISIONED
```

Broadcast Action: This broadcast is sent to indicate that provisioning of a managed profile has completed successfully.

The broadcast is limited to the primary profile, to the app specified in the provisioning intent with action `ACTION_PROVISION_MANAGED_PROFILE`.

This intent will contain the following extras

- `Intent.EXTRA_USER`, corresponds to the `UserHandle` of the managed profile.
- `EXTRA_PROVISIONING_ACCOUNT_TO_MIGRATE`, corresponds to the account requested to be migrated at provisioning time, if any.

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

ACTION_PROFILE_OWNER_CHANGED

```
public static final String ACTION_PROFILE_OWNER_CHANGED
```

Broadcast action: sent when the profile owner is set, changed or cleared. This broadcast is sent only to the user managed by the new profile owner.

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

ACTION_PROVISIONING_SUCCESSFUL

```
public static final String ACTION_PROVISIONING_SUCCESSFUL
```

Activity action: This activity action is sent to indicate that provisioning of a managed profile or managed device has completed successfully. It'll be sent at the same time as `DeviceAdminReceiver.ACTION_PROFILE_PROVISIONING_COMPLETE` broadcast but this will be delivered faster as it's an activity intent.

The intent is only sent to the new device or profile owner.

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

ACTION_PROVISION_MANAGED_DEVICE

```
public static final String ACTION_PROVISION_MANAGED_DEVICE
```

This constant was deprecated in API level 31.
to support `Build.VERSION_CODES.S` and later, admin apps must implement activities with intent filters for the `ACTION_GET_PROVISIONING_MODE` and `ACTION_ADMIN_POLICY_COMPLIANCE` intent actions; using `ACTION_PROVISION_MANAGED_DEVICE` to start provisioning will cause the provisioning to fail; to additionally support pre- `Build.VERSION_CODES.S`, admin apps must also continue to use this constant.

Activity action: Starts the provisioning flow which sets up a managed device. Must be started with `android.app.Activity.startActivityForResult(Intent,int)`.

During device owner provisioning a device admin app is set as the owner of the device. A device owner has full control over the device. The device owner can not be modified by the user.

A typical use case would be a device that is owned by a company, but used by either an employee or client.

An intent with this action can be sent only on an unprovisioned device. It is possible to check if provisioning is allowed or not by querying the method `isProvisioningAllowed(String)`.

The intent contains the following extras:

- `EXTRA_PROVISIONING_DEVICE_ADMIN_COMPONENT_NAME`
- `EXTRA_PROVISIONING_SKIP_ENCRYPTION`, optional
- `EXTRA_PROVISIONING_LEAVE_ALL_SYSTEM_APPS_ENABLED`, optional
- `EXTRA_PROVISIONING_ADMIN_EXTRAS_BUNDLE`, optional
- `EXTRA_PROVISIONING_LOGO_URI`, optional
- `EXTRA_PROVISIONING_DISCLAIMERS`, optional
- `EXTRA_PROVISIONING_SKIP_EDUCATION_SCREENS`, optional

When device owner provisioning has completed, an intent of the type `DeviceAdminReceiver.ACTION_PROFILE_PROVISIONING_COMPLETE` is broadcast to the device owner.

From version `Build.VERSION_CODES.O`, when device owner provisioning has completed, along with the above broadcast, activity intent `ACTION_PROVISIONING_SUCCESSFUL` will also be sent to the device owner.

If provisioning fails, the device is factory reset.

A result code of `Activity.RESULT_OK` implies that the synchronous part of the provisioning flow was successful, although this doesn't guarantee the full flow will succeed. Conversely a result code of `Activity.RESULT_CANCELED` implies that the user backed-out of provisioning, or some precondition for provisioning wasn't met.

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

ACTION_PROVISION_MANAGED_PROFILE

```
public static final String ACTION_PROVISION_MANAGED_PROFILE
```

Activity action: Starts the provisioning flow which sets up a managed profile.

It is possible to check if provisioning is allowed or not by querying the method `isProvisioningAllowed(String)`.

The intent may contain the following extras:

Extra		Supported Versions
`EXTRA_PROVISIONING_ACCOUNT_TO_MIGRATE`		
`EXTRA_PROVISIONING_SKIP_ENCRYPTION`		`Build.VERSION_CODES.N` +
`EXTRA_PROVISIONING_ADMIN_EXTRAS_BUNDLE`		
`EXTRA_PROVISIONING_LOGO_URI`		
`EXTRA_PROVISIONING_SKIP_USER_CONSENT`	**Can only be used by an existing device owner trying to create a managed profile**	
`EXTRA_PROVISIONING_KEEP_ACCOUNT_ON_MIGRATION`		
`EXTRA_PROVISIONING_DISCLAIMERS`		
`EXTRA_PROVISIONING_DEVICE_ADMIN_PACKAGE_NAME`	**Required if `EXTRA_PROVISIONING_DEVICE_ADMIN_COMPONENT_NAME` is not specified. Must match the package name of the calling application.**	`Build.VERSION_CODES.LOLLIPOP` +
`EXTRA_PROVISIONING_`	**Required if `EXTRA_PROVISIONING_DEVICE_ADMIN_PACKAGE_NAME` is not specified. Package name must match the package name of the calling application.**	`Build.VERSION_CODES.M` +

Extra		Supported Versions
DEVICE_ADMIN_COMPONENT_NAME		
EXTRA_PROVISIONING_ALLOW_OFFLINE	On Build.VERSION_CODES.TIRAMISU +, when set to true this will **force** offline provisioning instead of allowing it	

When managed provisioning has completed, broadcasts are sent to the application specified in the provisioning intent. The DeviceAdminReceiver.ACTION_PROFILE_PROVISIONING_COMPLETE broadcast is sent in the managed profile and the ACTION_MANAGED_PROFILE_PROVISIONED broadcast is sent in the primary profile.

From version Build.VERSION_CODES.O , when managed provisioning has completed, along with the above broadcast, activity intent ACTION_PROVISIONING_SUCCESSFUL will also be sent to the profile owner.

If provisioning fails, the managed profile is removed so the device returns to its previous state.

If launched with android.app.Activity.startActivityForResult(Intent,int) a result code of Activity.RESULT_OK indicates that the synchronous part of the provisioning flow was successful, although this doesn't guarantee the full flow will succeed. Conversely a result code of Activity.RESULT_CANCELED indicates that the user backed-out of provisioning or some precondition for provisioning wasn't met.

If a device policy management role holder updater is present on the device, an internet connection attempt must be made prior to launching this intent.

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

ACTION_SET_NEW_PARENT_PROFILE_PASSWORD

```
public static final String ACTION_SET_NEW_PARENT_PROFILE_PASSWORD
```

Activity action: have the user enter a new password for the parent profile. If the intent is launched from within a managed profile, this will trigger entering a new password for the parent of the profile. The caller can optionally set EXTRA_DEVICE_PASSWORD_REQUIREMENT_ONLY to only enforce device-wide password requirement. In all other cases the behaviour is identical to ACTION_SET_NEW_PASSWORD .

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

ACTION_SET_NEW_PASSWORD

```
public static final String ACTION_SET_NEW_PASSWORD
```

Activity action: have the user enter a new password.

For admin apps, this activity should be launched after using setPasswordQuality(ComponentName,int) , or setPasswordMinimumLength(ComponentName,int) to have the user enter a new password that meets the current

requirements. You can use `isActivePasswordSufficient()` to determine whether you need to have the user select a new password in order to meet the current constraints. Upon being resumed from this activity, you can check the new password characteristics to see if they are sufficient.

Non-admin apps can use `getPasswordComplexity()` to check the current screen lock complexity, and use this activity with extra `EXTRA_PASSWORD_COMPLEXITY` to suggest to users how complex the app wants the new screen lock to be. Note that both `getPasswordComplexity()` and the extra `EXTRA_PASSWORD_COMPLEXITY` require the calling app to have the permission `permission.REQUEST_PASSWORD_COMPLEXITY`.

If the intent is launched from within a managed profile with a profile owner built against `Build.VERSION_CODES.M` or before, this will trigger entering a new password for the parent of the profile. For all other cases it will trigger entering a new password for the user or profile it is launched from.

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

ACTION_START_ENCRYPTION

```
public static final String ACTION_START_ENCRYPTION
```

Activity action: begin the process of encrypting data on the device. This activity should be launched after using `setStorageEncryption(ComponentName, boolean)` to request encryption be activated. After resuming from this activity, use `getStorageEncryption(ComponentName)` to check encryption status. However, on some devices this activity may never return, as it may trigger a reboot and in some cases a complete data wipe of the device.

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

ACTION_SYSTEM_UPDATE_POLICY_CHANGED

```
public static final String ACTION_SYSTEM_UPDATE_POLICY_CHANGED
```

Broadcast action: notify that a new local system update policy has been set by the device owner. The new policy can be retrieved by `getSystemUpdatePolicy()`.

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

APP_FUNCTIONS_DISABLED

```
public static final int APP_FUNCTIONS_DISABLED
```

Indicates that `AppFunctionManager` is controlled and disabled by policy, i.e. no apps in the current user are allowed to expose app functions.

Constant Value: 1 (0x00000001)

APP_FUNCTIONS_DISABLED_CROSS_PROFILE

```
public static final int APP_FUNCTIONS_DISABLED_CROSS_PROFILE
```

Indicates that `AppFunctionManager` is controlled and disabled by a policy for cross profile interactions only, i.e. app functions exposed by apps in the current user can only be invoked within the same user.

This is different from `APP_FUNCTIONS_DISABLED` in that it only disables cross profile interactions (even if the caller has permissions required to interact across users). appfunctions can still be used within the a user profile boundary.

Constant Value: 2 (0x00000002)

APP_FUNCTIONS_NOT_CONTROLLED_BY_POLICY

```
public static final int APP_FUNCTIONS_NOT_CONTROLLED_BY_POLICY
```

Indicates that `AppFunctionManager` is not controlled by policy.

If no admin set this policy, it means appfunctions are enabled.

Constant Value: 0 (0x00000000)

AUTO_TIME_ZONE_DISABLED

```
public static final int AUTO_TIME_ZONE_DISABLED
```

Specifies the "disabled" auto time zone state.

Constant Value: 1 (0x00000001)

AUTO_TIME_ZONE_ENABLED

```
public static final int AUTO_TIME_ZONE_ENABLED
```

Specifies the "enabled" auto time zone state.

Constant Value: 2 (0x00000002)

AUTO_TIME_ZONE_NOT_CONTROLLED_BY_POLICY

```
public static final int AUTO_TIME_ZONE_NOT_CONTROLLED_BY_POLICY
```

Specifies that the auto time zone state is not controlled by device policy.

Constant Value: 0 (0x00000000)

CONTENT_PROTECTION_DISABLED

```
public static final int CONTENT_PROTECTION_DISABLED
```

Indicates that content protection is controlled and disabled by a policy (default).

Constant Value: 1 (0x00000001)

CONTENT_PROTECTION_ENABLED

```
public static final int CONTENT_PROTECTION_ENABLED
```

Indicates that content protection is controlled and enabled by a policy.

Constant Value: 2 (0x00000002)

CONTENT_PROTECTION_NOT_CONTROLLED_BY_POLICY

```
public static final int CONTENT_PROTECTION_NOT_CONTROLLED_BY_POLICY
```

Indicates that content protection is not controlled by policy, allowing user to choose.

Constant Value: 0 (0x00000000)

DELEGATION_CERT_INSTALL

```
public static final String DELEGATION_CERT_INSTALL
```

Delegation of certificate installation and management. This scope grants access to the `getInstalledCaCerts(ComponentName)`, `hasCaCertInstalled(ComponentName, byte)`, `installCaCert(ComponentName, byte)`, `uninstallCaCert(ComponentName, byte)`, `uninstallAllUserCaCerts(ComponentName)` and `installKeyPair(ComponentName, PrivateKey, Certificate, String)` APIs. This scope also grants the ability to read identifiers that the delegating device owner or profile owner can obtain. See `getEnrollmentSpecificId()`.

Constant Value: "delegation-cert-install"

DELEGATION_ENABLE_SYSTEM_APP

```
public static final String DELEGATION_ENABLE_SYSTEM_APP
```

Delegation for enabling system apps. This scope grants access to the `enableSystemApp(ComponentName, Intent)` API.

Constant Value: "delegation-enable-system-app"

DELEGATION_INSTALL_EXISTING_PACKAGE

```
public static final String DELEGATION_INSTALL_EXISTING_PACKAGE
```

Delegation for installing existing packages. This scope grants access to the `installExistingPackage(ComponentName, String)` API.

Constant Value: "delegation-install-existing-package"

DELEGATION_NETWORK_LOGGING

```
public static final String DELEGATION_NETWORK_LOGGING
```

Grants access to `setNetworkLoggingEnabled(ComponentName, boolean)`, `isNetworkLoggingEnabled(ComponentName)` and `retrieveNetworkLogs(ComponentName, long)`. Once granted the delegated app will start receiving DelegatedAdminReceiver.onNetworkLogsAvailable() callback, and Device owner or Profile Owner will no longer receive the DeviceAdminReceiver.onNetworkLogsAvailable() callback. There can be at most one app that has this delegation. If another app already had delegated network logging access, it will lose the delegation when a new app is delegated.

Device Owner can grant this access since Android 10. Profile Owner of a managed profile can grant this access since Android 12.

Constant Value: "delegation-network-logging"

ENCRYPTION_STATUS_ACTIVATING

```
public static final int ENCRYPTION_STATUS_ACTIVATING
```

This constant was deprecated in API level 34.
This result code has never actually been used, so there is no reason for apps to check for it.

Result code for `getStorageEncryptionStatus()`: indicating that encryption is not currently active, but is currently being activated.

Constant Value: 2 (0x00000002)

ENCRYPTION_STATUS_ACTIVE_DEFAULT_KEY

```
public static final int ENCRYPTION_STATUS_ACTIVE_DEFAULT_KEY
```

Result code for `getStorageEncryptionStatus()`: indicating that encryption is active, but the encryption key is not cryptographically protected by the user's credentials.

This value can only be returned on devices that use Full Disk Encryption. Support for Full Disk Encryption was entirely removed in API level 33, having been replaced by File Based Encryption. With File Based Encryption, each user's credential-encrypted storage is always cryptographically protected by the user's credentials.

Constant Value: 4 (0x00000004)

ENCRYPTION_STATUS_ACTIVE_PER_USER

```
public static final int ENCRYPTION_STATUS_ACTIVE_PER_USER
```

Result code for `getStorageEncryptionStatus()`: indicating that encryption is active and the encryption key is tied to the user or profile.

This value is only returned to apps targeting API level 24 and above. For apps targeting earlier API levels, `ENCRYPTION_STATUS_ACTIVE` is returned, even if the encryption key is specific to the user or profile.

Constant Value: 5 (0x00000005)

```
public static final String EXTRA_ADD_EXPLANATION
```

An optional CharSequence providing additional explanation for why the admin is being added.

Constant Value: "android.app.extra.ADD_EXPLANATION"

```
public static final String EXTRA_DEVICE_ADMIN
```

The ComponentName of the administrator component.

Constant Value: "android.app.extra.DEVICE_ADMIN"

```
public static final String EXTRA_DEVICE_PASSWORD_REQUIREMENT_ONLY
```

A boolean extra for `ACTION_SET_NEW_PARENT_PROFILE_PASSWORD` requesting that only device password requirement is enforced during the parent profile password enrolment flow.

Normally when enrolling password for the parent profile, both the device-wide password requirement (requirement set via `getParentProfileInstance(ComponentName)` instance) and the profile password requirement are enforced, if the profile currently does not have a separate work challenge. By setting this to `true`, profile password requirement is explicitly disregarded.

Constant Value: "android.app.extra.DEVICE_PASSWORD_REQUIREMENT_ONLY"

```
public static final String EXTRA_PROVISIONING_ACCOUNT_TO_MIGRATE
```

An `Account` extra holding the account to migrate during managed profile provisioning.

If the account supplied is present in the user, it will be copied, along with its credentials to the managed profile and removed from the user.

Constant Value: "android.app.extra.PROVISIONING_ACCOUNT_TO_MIGRATE"

```
public static final String EXTRA_PROVISIONING_ALLOWED_PROVISIONING_MODES
```

An `ArrayList` of `Integer` extra specifying the allowed provisioning modes.

This extra will be passed to the admin app's `ACTION_GET_PROVISIONING_MODE` activity, whose result intent must contain `EXTRA_PROVISIONING_MODE` set to one of the values in this array.

If the value set to `EXTRA_PROVISIONING_MODE` is not in the array, provisioning will fail.

Constant Value: "android.app.extra.PROVISIONING_ALLOWED_PROVISIONING_MODES"

```
public static final String EXTRA_PROVISIONING_ALLOW_OFFLINE
```

A boolean extra indicating whether offline provisioning should be used.

The default value is `false`.

Constant Value: "android.app.extra.PROVISIONING_ALLOW_OFFLINE"

```
public static final String EXTRA_PROVISIONING_DEVICE_ADMIN_COMPONENT_NAME
```

A ComponentName extra indicating the `device admin receiver` of the application that will be set as the Device Policy Controller.

If an application starts provisioning directly via an intent with action `ACTION_PROVISION_MANAGED_DEVICE` the package name of this component has to match the package name of the application that started provisioning.

This component is set as device owner and active admin when device owner provisioning is started by an intent with action `ACTION_PROVISION_MANAGED_DEVICE` or by an NFC message containing an NFC record with MIME type `MIME_TYPE_PROVISIONING_NFC`. For the NFC record, the component name must be flattened to a string, via `ComponentName.flattenToShortString()`.

Constant Value: "android.app.extra.PROVISIONING_DEVICE_ADMIN_COMPONENT_NAME"

```
public static final String EXTRA_PROVISIONING_DEVICE_ADMIN_MINIMUM_VERSION_CODE
```

An int extra holding a minimum required version code for the device admin package. If the device admin is already installed on the device, it will only be re-downloaded from `EXTRA_PROVISIONING_DEVICE_ADMIN_PACKAGE_DOWNLOAD_LOCATION` if the version of the installed package is less than this version code.

Use in an NFC record with `MIME_TYPE_PROVISIONING_NFC` that starts device owner provisioning via an NFC bump. It can also be used for QR code provisioning.

Constant Value: "android.app.extra.PROVISIONING_DEVICE_ADMIN_MINIMUM_VERSION_CODE"

```
public static final String EXTRA_PROVISIONING_DEVICE_ADMIN_PACKAGE_DOWNLOAD_COOKIE_HEADER
```

A String extra holding a http cookie header which should be used in the http request to the url specified in `EXTRA_PROVISIONING_DEVICE_ADMIN_PACKAGE_DOWNLOAD_LOCATION`.

Use in an NFC record with `MIME_TYPE_PROVISIONING_NFC` that starts device owner provisioning via an NFC bump. It can also be used for QR code provisioning.

Constant Value:
"android.app.extra.PROVISIONING_DEVICE_ADMIN_PACKAGE_DOWNLOAD_COOKIE_HEADER"

```
public static final String EXTRA_PROVISIONING_DEVICE_ADMIN_PACKAGE_DOWNLOAD_LOCATION
```

A String extra holding a url that specifies the download location of the device admin package. When not provided it is assumed that the device admin package is already installed.

Use in an NFC record with `MIME_TYPE_PROVISIONING_NFC` that starts device owner provisioning via an NFC bump. It can also be used for QR code provisioning.

Constant Value: "android.app.extra.PROVISIONING_DEVICE_ADMIN_PACKAGE_DOWNLOAD_LOCATION"

```
public static final String EXTRA_PROVISIONING_DEVICE_ADMIN_PACKAGE_NAME
```

This constant was deprecated in API level 23.
Use `EXTRA_PROVISIONING_DEVICE_ADMIN_COMPONENT_NAME`.

A String extra holding the package name of the application that will be set as Device Policy Controller.

When this extra is set, the application must have exactly one `device admin receiver`. This receiver will be set as the Device Policy Controller.

Constant Value: "android.app.extra.PROVISIONING_DEVICE_ADMIN_PACKAGE_NAME"

```
public static final String EXTRA_PROVISIONING_DISCLAIMER_HEADER
```

A String extra of localized disclaimer header.

The extra is typically the company name of mobile device management application (MDM) or the organization name.

`System apps` can also insert a disclaimer by declaring an application-level meta-data in `AndroidManifest.xml`.

For example:

```
<meta-data
    android:name="android.app.extra.PROVISIONING_DISCLAIMER_HEADER"
```

```
      android:resource="@string/disclaimer_header"
 />
```

This must be accompanied with another extra using the key `EXTRA_PROVISIONING_DISCLAIMER_CONTENT`.

Constant Value: "android.app.extra.PROVISIONING_DISCLAIMER_HEADER"

```
public static final String EXTRA_PROVISIONING_EMAIL_ADDRESS
```

This constant was deprecated in API level 26.
From `Build.VERSION_CODES.O`, never used while provisioning the device.

Constant Value: "android.app.extra.PROVISIONING_EMAIL_ADDRESS"

```
public static final String EXTRA_PROVISIONING_IMEI
```

A string extra holding the IMEI (International Mobile Equipment Identity) of the device.

Constant Value: "android.app.extra.PROVISIONING_IMEI"

```
public static final String EXTRA_PROVISIONING_KEEP_ACCOUNT_ON_MIGRATION
```

Boolean extra to indicate that the `migrated account` should be kept.

If it's set to `true`, the account will not be removed from the user after it is migrated to the newly created user or profile.

Defaults to `false`

Constant Value: "android.app.extra.PROVISIONING_KEEP_ACCOUNT_ON_MIGRATION"

```
public static final String EXTRA_PROVISIONING_KEEP_SCREEN_ON
```

This constant was deprecated in API level 34.
from `Build.VERSION_CODES.UPSIDE_DOWN_CAKE`, the flag wouldn't be functional. The screen is kept on throughout the provisioning flow.

A `boolean` flag that indicates whether the screen should be on throughout the provisioning flow.

This extra can either be passed as an extra to the `ACTION_PROVISION_MANAGED_PROFILE` intent, or it can be returned by the admin app when performing the admin-integrated provisioning flow as a result of the `ACTION_GET_PROVISIONING_MODE` activity.

Constant Value: "android.app.extra.PROVISIONING_KEEP_SCREEN_ON"

```
public static final String EXTRA_PROVISIONING_LEAVE_ALL_SYSTEM_APPS_ENABLED
```

A Boolean extra that can be used by the mobile device management application to skip the disabling of system apps during provisioning when set to `true`.

Use in an NFC record with `MIME_TYPE_PROVISIONING_NFC`, an intent with action `ACTION_PROVISION_MANAGED_PROFILE` that starts profile owner provisioning or set as an extra to the intent result of the `ACTION_GET_PROVISIONING_MODE` activity.

Constant Value: "android.app.extra.PROVISIONING_LEAVE_ALL_SYSTEM_APPS_ENABLED"

```
public static final String EXTRA_PROVISIONING_LOCALE
```

A String extra holding the `Locale` that the device will be set to. Format: xx_yy, where xx is the language code, and yy the country code.

Use only for device owner provisioning. This extra can be returned by the admin app when performing the admin-integrated provisioning flow as a result of the `ACTION_GET_PROVISIONING_MODE` activity.

Use in an NFC record with `MIME_TYPE_PROVISIONING_NFC` that starts device owner provisioning via an NFC bump. It can also be used for QR code provisioning.

Constant Value: "android.app.extra.PROVISIONING_LOCALE"

```
public static final String EXTRA_PROVISIONING_LOCAL_TIME
```

A Long extra holding the wall clock time (in milliseconds) to be set on the device's `AlarmManager`.

Use only for device owner provisioning. This extra can be returned by the admin app when performing the admin-integrated provisioning flow as a result of the `ACTION_GET_PROVISIONING_MODE` activity.

Use in an NFC record with `MIME_TYPE_PROVISIONING_NFC` that starts device owner provisioning via an NFC bump. It can also be used for QR code provisioning.

Constant Value: "android.app.extra.PROVISIONING_LOCAL_TIME"

```
public static final String EXTRA_PROVISIONING_LOGO_URI
```

This constant was deprecated in API level 33.
Logo customization is no longer supported in the provisioning flow.

A `Uri` extra pointing to a logo image. This image will be shown during the provisioning. If this extra is not passed, a default image will be shown.

The following URI schemes are accepted:

- content (`ContentResolver.SCHEME_CONTENT`)
- android.resource (`ContentResolver.SCHEME_ANDROID_RESOURCE`)

It is the responsibility of the caller to provide an image with a reasonable pixel density for the device.

If a content: URI is passed, the intent should also have the flag `Intent.FLAG_GRANT_READ_URI_PERMISSION` and the uri should be added to the `ClipData` of the intent.

Constant Value: "android.app.extra.PROVISIONING_LOGO_URI"

EXTRA_PROVISIONING_MAIN_COLOR

```
public static final String EXTRA_PROVISIONING_MAIN_COLOR
```

This constant was deprecated in API level 31.
Color customization is no longer supported in the provisioning flow.

A integer extra indicating the predominant color to show during the provisioning. Refer to `Color` for how the color is represented.

Use with `ACTION_PROVISION_MANAGED_PROFILE` or `ACTION_PROVISION_MANAGED_DEVICE`.

Constant Value: "android.app.extra.PROVISIONING_MAIN_COLOR"

```
public static final String EXTRA_PROVISIONING_SENSORS_PERMISSION_GRANT_OPT_OUT
```

A boolean extra indicating the admin of a fully-managed device opts out of controlling permission grants for sensor-related permissions, see `setPermissionGrantState(ComponentName,String,String,int)`. The default for this extra is `false` - by default, the admin of a fully-managed device has the ability to grant sensors-related permissions.

Use only for device owner provisioning. This extra can be returned by the admin app when performing the admin-integrated provisioning flow as a result of the `ACTION_GET_PROVISIONING_MODE` activity.

This extra may also be provided to the admin app via an intent extra for `ACTION_GET_PROVISIONING_MODE`.

Constant Value: "android.app.extra.PROVISIONING_SENSORS_PERMISSION_GRANT_OPT_OUT"

```
public static final String EXTRA_PROVISIONING_SERIAL_NUMBER
```

A string extra holding the serial number of the device.

Constant Value: "android.app.extra.PROVISIONING_SERIAL_NUMBER"

```
public static final String EXTRA_PROVISIONING_SHOULD_LAUNCH_RESULT_INTENT
```

A boolean extra that determines whether the provisioning flow should launch the resulting launch intent, if one is supplied by the device policy management role holder via `EXTRA_RESULT_LAUNCH_INTENT`. Default value is `false`.

If `true`, the resulting intent will be launched by the provisioning flow, if one is supplied by the device policy management role holder.

If `false`, the resulting intent will be returned as `EXTRA_RESULT_LAUNCH_INTENT` to the provisioning initiator, if one is supplied by the device manager role holder. It will be the responsibility of the provisioning initiator to launch this `Intent` after provisioning completes.

This extra is respected when provided via the provisioning intent actions such as `ACTION_PROVISION_MANAGED_PROFILE`.

Constant Value: "android.app.extra.PROVISIONING_SHOULD_LAUNCH_RESULT_INTENT"

```
public static final String EXTRA_PROVISIONING_SKIP_EDUCATION_SCREENS
```

A boolean extra indicating if the education screens from the provisioning flow should be skipped. If unspecified, defaults to `false`.

This extra can be set in the following ways:

- By the admin app when performing the admin-integrated provisioning flow as a result of the `ACTION_GET_PROVISIONING_MODE` activity
- For managed account enrollment

If the education screens are skipped, it is the admin application's responsibility to display its own user education screens.

Constant Value: "android.app.extra.PROVISIONING_SKIP_EDUCATION_SCREENS"

```
public static final String EXTRA_PROVISIONING_SKIP_USER_CONSENT
```

This constant was deprecated in API level 31.
this extra is no longer relevant as device owners cannot create managed profiles

A boolean extra indicating if the user consent steps from the provisioning flow should be skipped.

If unspecified, defaults to `false`.

Constant Value: "android.app.extra.PROVISIONING_SKIP_USER_CONSENT"

```
public static final String EXTRA_PROVISIONING_TIME_ZONE
```

A String extra holding the time zone `AlarmManager` that the device will be set to.

Use only for device owner provisioning. This extra can be returned by the admin app when performing the admin-integrated provisioning flow as a result of the `ACTION_GET_PROVISIONING_MODE` activity.

Use in an NFC record with `MIME_TYPE_PROVISIONING_NFC` that starts device owner provisioning via an NFC bump. It can also be used for QR code provisioning.

Constant Value: "android.app.extra.PROVISIONING_TIME_ZONE"

```
public static final String EXTRA_PROVISIONING_USE_MOBILE_DATA
```

A boolean extra indicating if mobile data should be used during the provisioning flow for downloading the admin app. If `EXTRA_PROVISIONING_WIFI_SSID` is also specified, wifi network will be used instead.

Default value is `false`.

If this extra is set to `true` and `EXTRA_PROVISIONING_WIFI_SSID` is not specified, this extra has different behaviour depending on the way provisioning is triggered:

- For provisioning started via a QR code or an NFC tag, mobile data is always used for downloading the admin app.
- For all other provisioning methods, a mobile data connection check is made at the start of provisioning. If mobile data is connected at that point, the admin app download will happen using mobile data. If mobile data is not connected at that point, the end-user will be asked to pick a wifi network and the admin app download will proceed over wifi.

Constant Value: "android.app.extra.PROVISIONING_USE_MOBILE_DATA"

```
public static final String EXTRA_PROVISIONING_WIFI_CA_CERTIFICATE
```

The CA certificate of the wifi network in `EXTRA_PROVISIONING_WIFI_SSID`. This should be an X.509 certificate Base64 encoded DER format, ie. PEM representation of a certificate without header, footer and line breaks. More information This is only used if the `EXTRA_PROVISIONING_WIFI_SECURITY_TYPE` is `EAP`.

Use in an NFC record with `MIME_TYPE_PROVISIONING_NFC` that starts device owner provisioning via an NFC bump. It can also be used for QR code provisioning.

Constant Value: "android.app.extra.PROVISIONING_WIFI_CA_CERTIFICATE"

```
public static final String EXTRA_PROVISIONING_WIFI_HIDDEN
```

A boolean extra indicating whether the wifi network in `EXTRA_PROVISIONING_WIFI_SSID` is hidden or not.

Use in an NFC record with `MIME_TYPE_PROVISIONING_NFC` that starts device owner provisioning via an NFC bump. It can also be used for QR code provisioning.

Constant Value: "android.app.extra.PROVISIONING_WIFI_HIDDEN"

```
public static final String EXTRA_PROVISIONING_WIFI_PAC_URL
```

A String extra holding the proxy auto-config (PAC) URL for the wifi network in `EXTRA_PROVISIONING_WIFI_SSID`.

Use in an NFC record with `MIME_TYPE_PROVISIONING_NFC` that starts device owner provisioning via an NFC bump. It can also be used for QR code provisioning.

Constant Value: "android.app.extra.PROVISIONING_WIFI_PAC_URL"

```
public static final String EXTRA_PROVISIONING_WIFI_PASSWORD
```

A String extra holding the password of the wifi network in `EXTRA_PROVISIONING_WIFI_SSID`.

Use in an NFC record with `MIME_TYPE_PROVISIONING_NFC` that starts device owner provisioning via an NFC bump. It can also be used for QR code provisioning.

Constant Value: "android.app.extra.PROVISIONING_WIFI_PASSWORD"

```
public static final String EXTRA_PROVISIONING_WIFI_PHASE2_AUTH
```

The phase 2 authentication of the wifi network in `EXTRA_PROVISIONING_WIFI_SSID` and could be one of `NONE`, `PAP`, `MSCHAP`, `MSCHAPV2`, `GTC`, `SIM`, `AKA` or `AKA_PRIME`. This is only used if the `EXTRA_PROVISIONING_WIFI_SECURITY_TYPE` is `EAP`.

Use in an NFC record with `MIME_TYPE_PROVISIONING_NFC` that starts device owner provisioning via an NFC bump. It can also be used for QR code provisioning.

Constant Value: "android.app.extra.PROVISIONING_WIFI_PHASE2_AUTH"

```
public static final String EXTRA_PROVISIONING_WIFI_PROXY_BYPASS
```

A String extra holding the proxy bypass for the wifi network in `EXTRA_PROVISIONING_WIFI_SSID`.

Use in an NFC record with `MIME_TYPE_PROVISIONING_NFC` that starts device owner provisioning via an NFC bump. It can also be used for QR code provisioning.

Constant Value: "android.app.extra.PROVISIONING_WIFI_PROXY_BYPASS"

```
public static final String EXTRA_PROVISIONING_WIFI_PROXY_HOST
```

A String extra holding the proxy host for the wifi network in `EXTRA_PROVISIONING_WIFI_SSID`.

Use in an NFC record with `MIME_TYPE_PROVISIONING_NFC` that starts device owner provisioning via an NFC bump. It can also be used for QR code provisioning.

Constant Value: "android.app.extra.PROVISIONING_WIFI_PROXY_HOST"

```
public static final String EXTRA_PROVISIONING_WIFI_PROXY_PORT
```

An int extra holding the proxy port for the wifi network in `EXTRA_PROVISIONING_WIFI_SSID`.

Use in an NFC record with `MIME_TYPE_PROVISIONING_NFC` that starts device owner provisioning via an NFC bump. It can also be used for QR code provisioning.

Constant Value: "android.app.extra.PROVISIONING_WIFI_PROXY_PORT"

```
public static final String EXTRA_PROVISIONING_WIFI_SECURITY_TYPE
```

A String extra indicating the security type of the wifi network in `EXTRA_PROVISIONING_WIFI_SSID` and could be one of `NONE` , `WPA` , `WEP` or `EAP` .

Use in an NFC record with `MIME_TYPE_PROVISIONING_NFC` that starts device owner provisioning via an NFC bump. It can also be used for QR code provisioning.

Constant Value: "android.app.extra.PROVISIONING_WIFI_SECURITY_TYPE"

```
public static final String EXTRA_PROVISIONING_WIFI_SSID
```

A String extra holding the ssid of the wifi network that should be used during nfc device owner provisioning for downloading the mobile device management application.

Use in an NFC record with `MIME_TYPE_PROVISIONING_NFC` that starts device owner provisioning via an NFC bump. It can also be used for QR code provisioning.

Constant Value: "android.app.extra.PROVISIONING_WIFI_SSID"

```
public static final String EXTRA_PROVISIONING_WIFI_USER_CERTIFICATE
```

The user certificate of the wifi network in `EXTRA_PROVISIONING_WIFI_SSID` . This should be an X.509 certificate and private key Base64 encoded DER format, ie. PEM representation of a certificate and key without header, footer and line breaks. More information This is only used if the `EXTRA_PROVISIONING_WIFI_SECURITY_TYPE` is `EAP` .

Use in an NFC record with `MIME_TYPE_PROVISIONING_NFC` that starts device owner provisioning via an NFC bump. It can also be used for QR code provisioning.

Constant Value: "android.app.extra.PROVISIONING_WIFI_USER_CERTIFICATE"

```
public static final String EXTRA_RESOURCE_IDS
```

An integer array extra for `ACTION_DEVICE_POLICY_RESOURCE_UPDATED` to indicate which resource IDs (i.e. strings and drawables) have been updated.

Constant Value: "android.app.extra.RESOURCE_IDS"

```
public static final int EXTRA_RESOURCE_TYPE_DRAWABLE
```

A `int` value for `EXTRA_RESOURCE_TYPE` to indicate that a resource of type `Drawable` is being updated.

Constant Value: 1 (0x00000001)

```
public static final int EXTRA_RESOURCE_TYPE_STRING
```

A `int` value for `EXTRA_RESOURCE_TYPE` to indicate that a resource of type `String` is being updated.

Constant Value: 2 (0x00000002)

FLAG_EVICT_CREDENTIAL_ENCRYPTION_KEY

```
public static final int FLAG_EVICT_CREDENTIAL_ENCRYPTION_KEY
```

Flag for `lockNow(int)` : also evict the user's credential encryption key from the keyring. The user's credential will need to be entered again in order to derive the credential encryption key that will be stored back in the keyring for future use.

This flag can only be used by a profile owner when locking a managed profile when `getStorageEncryptionStatus()` returns `ENCRYPTION_STATUS_ACTIVE_PER_USER` .

In order to secure user data, the user will be stopped and restarted so apps should wait until they are next run to perform further actions.

Constant Value: 1 (0x00000001)

INSTALLKEY_REQUEST_CREDENTIALS_ACCESS

```
public static final int INSTALLKEY_REQUEST_CREDENTIALS_ACCESS
```

Specifies that the calling app should be granted access to the installed credentials immediately. Otherwise, access to the credentials will be gated by user approval. For use with `installKeyPair(ComponentName,PrivateKey,Certificate[],String,int)`

Constant Value: 1 (0x00000001)

INSTALLKEY_SET_USER_SELECTABLE

```
public static final int INSTALLKEY_SET_USER_SELECTABLE
```

Specifies that a user can select the key via the Certificate Selection prompt. If this flag is not set when calling `installKeyPair(ComponentName, PrivateKey, Certificate, String)` , the key can only be granted access by implementing `DeviceAdminReceiver.onChoosePrivateKeyAlias(Context, Intent, int, Uri, String)` . For use with `installKeyPair(ComponentName,PrivateKey,Certificate[],String,int)`

Constant Value: 2 (0x00000002)

KEYGUARD_DISABLE_BIOMETRICS

```
public static final int KEYGUARD_DISABLE_BIOMETRICS
```

Disable all biometric authentication on keyguard secure screens (e.g. PIN/Pattern/Password).

Constant Value: 416 (0x000001a0)

KEYGUARD_DISABLE_FACE

```
public static final int KEYGUARD_DISABLE_FACE
```

Disable face authentication on keyguard secure screens (e.g. PIN/Pattern/Password).

Constant Value: 128 (0x00000080)

KEYGUARD_DISABLE_FEATURES_ALL

```
public static final int KEYGUARD_DISABLE_FEATURES_ALL
```

Disable all current and future keyguard customizations.

Constant Value: 2147483647 (0x7fffffff)

KEYGUARD_DISABLE_FEATURES_NONE

```
public static final int KEYGUARD_DISABLE_FEATURES_NONE
```

Widgets are enabled in keyguard

Constant Value: 0 (0x00000000)

KEYGUARD_DISABLE_FINGERPRINT

```
public static final int KEYGUARD_DISABLE_FINGERPRINT
```

Disable fingerprint authentication on keyguard secure screens (e.g. PIN/Pattern/Password).

Constant Value: 32 (0x00000020)

KEYGUARD_DISABLE_IRIS

```
public static final int KEYGUARD_DISABLE_IRIS
```

Disable iris authentication on keyguard secure screens (e.g. PIN/Pattern/Password).

Constant Value: 256 (0x00000100)

KEYGUARD_DISABLE_REMOTE_INPUT

```
public static final int KEYGUARD_DISABLE_REMOTE_INPUT
```

This constant was deprecated in API level 33.
This flag was added in version `Build.VERSION_CODES.N`, but it never had any effect.

Disable text entry into notifications on secure keyguard screens (e.g. PIN/Pattern/Password).

Constant Value: 64 (0x00000040)

KEYGUARD_DISABLE_SECURE_CAMERA

```
public static final int KEYGUARD_DISABLE_SECURE_CAMERA
```

Disable the camera on secure keyguard screens (e.g. PIN/Pattern/Password)

Constant Value: 2 (0x00000002)

KEYGUARD_DISABLE_SECURE_NOTIFICATIONS

```
public static final int KEYGUARD_DISABLE_SECURE_NOTIFICATIONS
```

Disable showing all notifications on secure keyguard screens (e.g. PIN/Pattern/Password)

Constant Value: 4 (0x00000004)

KEYGUARD_DISABLE_SHORTCUTS_ALL

```
public static final int KEYGUARD_DISABLE_SHORTCUTS_ALL
```

Disable all keyguard shortcuts.

Constant Value: 512 (0x00000200)

KEYGUARD_DISABLE_TRUST_AGENTS

```
public static final int KEYGUARD_DISABLE_TRUST_AGENTS
```

Disable trust agents on secure keyguard screens (e.g. PIN/Pattern/Password). By setting this flag alone, all trust agents are disabled. If the admin then wants to allowlist specific features of some trust agent, `setTrustAgentConfiguration(ComponentName, ComponentName, PersistableBundle)` can be used in conjuction to set trust-agent-specific configurations.

Constant Value: 16 (0x00000010)

KEYGUARD_DISABLE_UNREDACTED_NOTIFICATIONS

```
public static final int KEYGUARD_DISABLE_UNREDACTED_NOTIFICATIONS
```

Only allow redacted notifications on secure keyguard screens (e.g. PIN/Pattern/Password)

Constant Value: 8 (0x00000008)

KEYGUARD_DISABLE_WIDGETS_ALL

```
public static final int KEYGUARD_DISABLE_WIDGETS_ALL
```

Disable all keyguard widgets. Has no effect between `Build.VERSION_CODES.LOLLIPOP` and `Build.VERSION_CODES.UPSIDE_DOWN_CAKE` (both inclusive), since keyguard widget is only supported on Android versions lower than 5.0 and versions higher than 14.

Constant Value: 1 (0x00000001)

LOCK_TASK_FEATURE_BLOCK_ACTIVITY_START_IN_TASK

```
public static final int LOCK_TASK_FEATURE_BLOCK_ACTIVITY_START_IN_TASK
```

Enable blocking of non-allowlisted activities from being started into a locked task.

Constant Value: 64 (0x00000040)

LOCK_TASK_FEATURE_GLOBAL_ACTIONS

```
public static final int LOCK_TASK_FEATURE_GLOBAL_ACTIONS
```

Enable the global actions dialog during LockTask mode. This is the dialog that shows up when the user long-presses the power button, for example. Note that the user may not be able to power off the device if this flag is not set.

This flag is enabled by default until `setLockTaskFeatures(ComponentName,int)` is called for the first time.

Constant Value: 16 (0x00000010)

LOCK_TASK_FEATURE_KEYGUARD

```
public static final int LOCK_TASK_FEATURE_KEYGUARD
```

Enable the keyguard during LockTask mode. Note that if the keyguard is already disabled with `setKeyguardDisabled(ComponentName,boolean)`, setting this flag will have no effect. If this flag is not set, the keyguard will not be shown even if the user has a lock screen credential.

Constant Value: 32 (0x00000020)

LOCK_TASK_FEATURE_NONE

```
public static final int LOCK_TASK_FEATURE_NONE
```

Disable all configurable SystemUI features during LockTask mode. This includes,

- system info area in the status bar (connectivity icons, clock, etc.)
- notifications (including alerts, icons, and the notification shade)
- Home button
- Recents button and UI
- global actions menu (i.e. power button menu)
- keyguard

Constant Value: 0 (0x00000000)

LOCK_TASK_FEATURE_NOTIFICATIONS

```
public static final int LOCK_TASK_FEATURE_NOTIFICATIONS
```

Enable notifications during LockTask mode. This includes notification icons on the status bar, heads-up notifications, and the expandable notification shade. Note that the Quick Settings panel remains disabled. This feature flag can only be used in combination with `LOCK_TASK_FEATURE_HOME`. `setLockTaskFeatures(ComponentName,int)` throws an `IllegalArgumentException` if this feature flag is defined without `LOCK_TASK_FEATURE_HOME`.

Constant Value: 2 (0x00000002)

LOCK_TASK_FEATURE_SYSTEM_INFO

```
public static final int LOCK_TASK_FEATURE_SYSTEM_INFO
```

Enable the system info area in the status bar during LockTask mode. The system info area usually occupies the right side of the status bar (although this can differ across OEMs). It includes all system information indicators, such as date and time, connectivity, battery, vibration mode, etc.

Constant Value: 1 (0x00000001)

MIME_TYPE_PROVISIONING_NFC

```
public static final String MIME_TYPE_PROVISIONING_NFC
```

This MIME type is used for starting the device owner provisioning.

During device owner provisioning a device admin app is set as the owner of the device. A device owner has full control over the device. The device owner can not be modified by the user and the only way of resetting the device is if the device owner app calls a factory reset.

A typical use case would be a device that is owned by a company, but used by either an employee or client.

The NFC message must be sent to an unprovisioned device.

The NFC record must contain a serialized `Properties` object which contains the following properties:

- `EXTRA_PROVISIONING_DEVICE_ADMIN_PACKAGE_NAME`
- `EXTRA_PROVISIONING_DEVICE_ADMIN_PACKAGE_DOWNLOAD_LOCATION` , optional
- `EXTRA_PROVISIONING_DEVICE_ADMIN_PACKAGE_DOWNLOAD_COOKIE_HEADER` , optional
- `EXTRA_PROVISIONING_DEVICE_ADMIN_PACKAGE_CHECKSUM` , optional
- `EXTRA_PROVISIONING_LOCAL_TIME` (convert to String), optional
- `EXTRA_PROVISIONING_TIME_ZONE` , optional
- `EXTRA_PROVISIONING_LOCALE` , optional
- `EXTRA_PROVISIONING_WIFI_SSID` , optional
- `EXTRA_PROVISIONING_WIFI_HIDDEN` (convert to String), optional
- `EXTRA_PROVISIONING_WIFI_SECURITY_TYPE` , optional
- `EXTRA_PROVISIONING_WIFI_PASSWORD` , optional
- `EXTRA_PROVISIONING_WIFI_PROXY_HOST` , optional
- `EXTRA_PROVISIONING_WIFI_PROXY_PORT` (convert to String), optional
- `EXTRA_PROVISIONING_WIFI_PROXY_BYPASS` , optional
- `EXTRA_PROVISIONING_WIFI_PAC_URL` , optional
- `EXTRA_PROVISIONING_ADMIN_EXTRAS_BUNDLE` , optional, supported from `Build.VERSION_CODES.M`
- `EXTRA_PROVISIONING_WIFI_EAP_METHOD` , optional, supported from `Build.VERSION_CODES.Q`
- `EXTRA_PROVISIONING_WIFI_PHASE2_AUTH` , optional, supported from `Build.VERSION_CODES.Q`
- `EXTRA_PROVISIONING_WIFI_CA_CERTIFICATE` , optional, supported from `Build.VERSION_CODES.Q`
- `EXTRA_PROVISIONING_WIFI_USER_CERTIFICATE` , optional, supported from `Build.VERSION_CODES.Q`
- `EXTRA_PROVISIONING_WIFI_IDENTITY` , optional, supported from `Build.VERSION_CODES.Q`
- `EXTRA_PROVISIONING_WIFI_ANONYMOUS_IDENTITY` , optional, supported from `Build.VERSION_CODES.Q`
- `EXTRA_PROVISIONING_WIFI_DOMAIN` , optional, supported from `Build.VERSION_CODES.Q`

As of `Build.VERSION_CODES.M` , the properties should contain `EXTRA_PROVISIONING_DEVICE_ADMIN_COMPONENT_NAME` instead of `EXTRA_PROVISIONING_DEVICE_ADMIN_PACKAGE_NAME` , (although specifying only `EXTRA_PROVISIONING_DEVICE_ADMIN_PACKAGE_NAME` is still supported).

Constant Value: "application/com.android.managedprovisioning"

MTE_DISABLED

```
public static final int MTE_DISABLED
```

Require that MTE be disabled on the device. Can be set by a device owner.

Constant Value: 2 (0x00000002)

MTE_ENABLED

```
public static final int MTE_ENABLED
```

Require that MTE be enabled on the device, if supported. Can be set by a device owner or a profile owner of an organization-owned managed profile.

Constant Value: 1 (0x00000001)

MTE_NOT_CONTROLLED_BY_POLICY

```
public static final int MTE_NOT_CONTROLLED_BY_POLICY
```

Allow the user to choose whether to enable MTE on the device.

Constant Value: 0 (0x00000000)

NEARBY_STREAMING_DISABLED

```
public static final int NEARBY_STREAMING_DISABLED
```

Indicates that nearby streaming is disabled.

Constant Value: 1 (0x00000001)

NEARBY_STREAMING_ENABLED

```
public static final int NEARBY_STREAMING_ENABLED
```

Indicates that nearby streaming is enabled.

Constant Value: 2 (0x00000002)

NEARBY_STREAMING_NOT_CONTROLLED_BY_POLICY

```
public static final int NEARBY_STREAMING_NOT_CONTROLLED_BY_POLICY
```

Indicates that nearby streaming is not controlled by policy, which means nearby streaming is allowed.

Constant Value: 0 (0x00000000)

NEARBY_STREAMING_SAME_MANAGED_ACCOUNT_ONLY

```
public static final int NEARBY_STREAMING_SAME_MANAGED_ACCOUNT_ONLY
```

Indicates that nearby streaming is enabled only to devices offering a comparable level of security, with the same authenticated managed account.

Constant Value: 3 (0x00000003)

OPERATION_SAFETY_REASON_DRIVING_DISTRACTION

```
public static final int OPERATION_SAFETY_REASON_DRIVING_DISTRACTION
```

Indicates that a `UnsafeStateException` was thrown because the operation would distract the driver of the vehicle.

Constant Value: 1 (0x00000001)

PASSWORD_COMPLEXITY_HIGH

```
public static final int PASSWORD_COMPLEXITY_HIGH
```

Constant for `getPasswordComplexity()` and `setRequiredPasswordComplexity(int)`. Define the high password complexity band as:

- PIN with **no** repeating (4444) or ordered (1234, 4321, 2468) sequences, length at least 8
- alphabetic, length at least 6
- alphanumeric, length at least 6

When returned from `getPasswordComplexity()`, the constant represents the exact complexity band the password is in. When passed to `setRequiredPasswordComplexity(int)`, it sets the minimum complexity band which the password must meet.

Constant Value: 327680 (0x00050000)

PASSWORD_COMPLEXITY_MEDIUM

```
public static final int PASSWORD_COMPLEXITY_MEDIUM
```

Constant for `getPasswordComplexity()` and `setRequiredPasswordComplexity(int)`. Define the medium password complexity band as:

- PIN with **no** repeating (4444) or ordered (1234, 4321, 2468) sequences, length at least 4
- alphabetic, length at least 4
- alphanumeric, length at least 4

When returned from `getPasswordComplexity()`, the constant represents the exact complexity band the password is in. When passed to `setRequiredPasswordComplexity(int)`, it sets the minimum complexity band which the password must meet.

Constant Value: 196608 (0x00030000)

PASSWORD_QUALITY_ALPHABETIC

```
public static final int PASSWORD_QUALITY_ALPHABETIC
```

Constant for `setPasswordQuality(ComponentName, int)` : the user must have entered a password containing at least alphabetic (or other symbol) characters. Note that quality constants are ordered so that higher values are more restrictive.

Constant Value: 262144 (0x00040000)

PASSWORD_QUALITY_ALPHANUMERIC

```
public static final int PASSWORD_QUALITY_ALPHANUMERIC
```

Constant for `setPasswordQuality(ComponentName, int)` : the user must have entered a password containing at least *both*> numeric *and* alphabetic (or other symbol) characters. Note that quality constants are ordered so that higher values are more restrictive.

Constant Value: 327680 (0x00050000)

PASSWORD_QUALITY_BIOMETRIC_WEAK

```
public static final int PASSWORD_QUALITY_BIOMETRIC_WEAK
```

Constant for `setPasswordQuality(ComponentName, int)` : the policy allows for low-security biometric recognition technology. This implies technologies that can recognize the identity of an individual to about a 3 digit PIN (false detection is less than 1 in 1,000). Note that quality constants are ordered so that higher values are more restrictive.

Constant Value: 32768 (0x00008000)

PASSWORD_QUALITY_COMPLEX

```
public static final int PASSWORD_QUALITY_COMPLEX
```

Constant for `setPasswordQuality(ComponentName, int)` : allows the admin to set precisely how many characters of various types the password should contain to satisfy the policy. The admin should set these requirements via `setPasswordMinimumLetters(ComponentName, int)` , `setPasswordMinimumNumeric(ComponentName, int)` , `setPasswordMinimumSymbols(ComponentName, int)` , `setPasswordMinimumUpperCase(ComponentName, int)` , `setPasswordMinimumLowerCase(ComponentName, int)` , `setPasswordMinimumNonLetter(ComponentName, int)` , and `setPasswordMinimumLength(ComponentName, int)` . Note that quality constants are ordered so that higher values are more restrictive.

Constant Value: 393216 (0x00060000)

PASSWORD_QUALITY_NUMERIC

```
public static final int PASSWORD_QUALITY_NUMERIC
```

Constant for `setPasswordQuality(ComponentName, int)` : the user must have entered a password containing at least numeric characters. Note that quality constants are ordered so that higher values are more restrictive.

Constant Value: 131072 (0x00020000)

PASSWORD_QUALITY_NUMERIC_COMPLEX

```
public static final int PASSWORD_QUALITY_NUMERIC_COMPLEX
```

Constant for `setPasswordQuality(ComponentName, int)` : the user must have entered a password containing at least numeric characters with no repeating (4444) or ordered (1234, 4321, 2468) sequences. Note that quality constants are ordered so that higher values are more restrictive.

Constant Value: 196608 (0x00030000)

PASSWORD_QUALITY_SOMETHING

```
public static final int PASSWORD_QUALITY_SOMETHING
```

Constant for `setPasswordQuality(ComponentName, int)` : the policy requires some kind of password or pattern, but doesn't care what it is. Note that quality constants are ordered so that higher values are more restrictive.

Constant Value: 65536 (0x00010000)

PASSWORD_QUALITY_UNSPECIFIED

```
public static final int PASSWORD_QUALITY_UNSPECIFIED
```

Constant for `setPasswordQuality(ComponentName, int)` : the policy has no requirements for the password. Note that quality constants are ordered so that higher values are more restrictive.

Constant Value: 0 (0x00000000)

PERMISSION_GRANT_STATE_DEFAULT

```
public static final int PERMISSION_GRANT_STATE_DEFAULT
```

Runtime permission state: The user can manage the permission through the UI.

Constant Value: 0 (0x00000000)

PERMISSION_GRANT_STATE_DENIED

```
public static final int PERMISSION_GRANT_STATE_DENIED
```

Runtime permission state: The permission is denied to the app and the user cannot manage the permission through the UI.

Constant Value: 2 (0x00000002)

PERMISSION_GRANT_STATE_GRANTED

```
public static final int PERMISSION_GRANT_STATE_GRANTED
```

Runtime permission state: The permission is granted to the app and the user cannot manage the permission through the UI.

Constant Value: 1 (0x00000001)

PERMISSION_POLICY_AUTO_DENY

```
public static final int PERMISSION_POLICY_AUTO_DENY
```

Permission policy to always deny new permission requests for runtime permissions. Already granted or denied permissions are not affected by this.

Constant Value: 2 (0x00000002)

PERMISSION_POLICY_AUTO_GRANT

```
public static final int PERMISSION_POLICY_AUTO_GRANT
```

Permission policy to always grant new permission requests for runtime permissions. Already granted or denied permissions are not affected by this.

Constant Value: 1 (0x00000001)

PERMISSION_POLICY_PROMPT

```
public static final int PERMISSION_POLICY_PROMPT
```

Permission policy to prompt user for new permission requests for runtime permissions. Already granted or denied permissions are not affected by this.

Constant Value: 0 (0x00000000)

PERSONAL_APPS_SUSPENDED_PROFILE_TIMEOUT

```
public static final int PERSONAL_APPS_SUSPENDED_PROFILE_TIMEOUT
```

Flag for `getPersonalAppsSuspendedReasons(ComponentName)` return value. Set when personal apps are suspended by framework because managed profile was off for longer than allowed by policy.

Constant Value: 2 (0x00000002)

POLICY_DISABLE_CAMERA

```
public static final String POLICY_DISABLE_CAMERA
```

Constant to indicate the feature of disabling the camera. Used as argument to `createAdminSupportIntent(String)`.

Constant Value: "policy_disable_camera"

POLICY_DISABLE_SCREEN_CAPTURE

```
public static final String POLICY_DISABLE_SCREEN_CAPTURE
```

Constant to indicate the feature of disabling screen captures. Used as argument to `createAdminSupportIntent(String)`.

Constant Value: "policy_disable_screen_capture"

PRIVATE_DNS_MODE_OFF

```
public static final int PRIVATE_DNS_MODE_OFF
```

Specifies that Private DNS was turned off completely.

Constant Value: 1 (0x00000001)

PRIVATE_DNS_MODE_OPPORTUNISTIC

```
public static final int PRIVATE_DNS_MODE_OPPORTUNISTIC
```

Specifies that the device owner requested opportunistic DNS over TLS

Constant Value: 2 (0x00000002)

PRIVATE_DNS_MODE_PROVIDER_HOSTNAME

```
public static final int PRIVATE_DNS_MODE_PROVIDER_HOSTNAME
```

Specifies that the device owner configured a specific host to use for Private DNS.

Constant Value: 3 (0x00000003)

PRIVATE_DNS_MODE_UNKNOWN

```
public static final int PRIVATE_DNS_MODE_UNKNOWN
```

Specifies that the Private DNS setting is in an unknown state.

Constant Value: 0 (0x00000000)

PRIVATE_DNS_SET_ERROR_FAILURE_SETTING

```
public static final int PRIVATE_DNS_SET_ERROR_FAILURE_SETTING
```

General failure to set the Private DNS mode, not due to one of the reasons listed above.

Constant Value: 2 (0x00000002)

PRIVATE_DNS_SET_ERROR_HOST_NOT_SERVING

```
public static final int PRIVATE_DNS_SET_ERROR_HOST_NOT_SERVING
```

If the `privateDnsHost` provided was of a valid hostname but that host was found to not support DNS-over-TLS.

Constant Value: 1 (0x00000001)

PRIVATE_DNS_SET_NO_ERROR

```
public static final int PRIVATE_DNS_SET_NO_ERROR
```

The selected mode has been set successfully. If the mode is `PRIVATE_DNS_MODE_PROVIDER_HOSTNAME` then it implies the supplied host is valid and reachable.

Constant Value: 0 (0x00000000)

PROVISIONING_MODE_FULLY_MANAGED_DEVICE

```
public static final int PROVISIONING_MODE_FULLY_MANAGED_DEVICE
```

The provisioning mode for fully managed device.

Constant Value: 1 (0x00000001)

PROVISIONING_MODE_MANAGED_PROFILE

```
public static final int PROVISIONING_MODE_MANAGED_PROFILE
```

The provisioning mode for managed profile.

Constant Value: 2 (0x00000002)

PROVISIONING_MODE_MANAGED_PROFILE_ON_PERSONAL_DEVICE

```
public static final int PROVISIONING_MODE_MANAGED_PROFILE_ON_PERSONAL_DEVICE
```

The provisioning mode for a managed profile on a personal device.

This mode is only available when the provisioning initiator has explicitly instructed the provisioning flow to support managed profile on a personal device provisioning. In that case, `PROVISIONING_MODE_MANAGED_PROFILE` corresponds to an organization-owned managed profile, whereas this constant corresponds to a personally-owned managed profile.

Constant Value: 3 (0x00000003)

RESET_PASSWORD_DO_NOT_ASK_CREDENTIALS_ON_BOOT

```
public static final int RESET_PASSWORD_DO_NOT_ASK_CREDENTIALS_ON_BOOT
```

Flag for `resetPasswordWithToken(ComponentName, String, byte, int)` and `resetPassword(String, int)` : don't ask for user credentials on device boot. If the flag is set, the device can be booted without asking for user password. The absence of this flag does not change the current boot requirements. This flag can be set by the device owner only. If the app is not the device owner, the flag is ignored. Once the flag is set, it cannot be reverted back without resetting the device to factory defaults.

Constant Value: 2 (0x00000002)

WIPE_EUICC

```
public static final int WIPE_EUICC
```

Flag for `wipeDevice(int)` : also erase the device's eUICC data.

Constant Value: 4 (0x00000004)

WIPE_EXTERNAL_STORAGE

```
public static final int WIPE_EXTERNAL_STORAGE
```

Flag for `wipeData(int)` : also erase the device's adopted external storage (such as adopted SD cards).

Constant Value: 1 (0x00000001)

WIPE_RESET_PROTECTION_DATA

```
public static final int WIPE_RESET_PROTECTION_DATA
```

Flag for `wipeData(int)` : also erase the factory reset protection data.

This flag may only be set by device owner admins; if it is set by other admins a `SecurityException` will be thrown.

Constant Value: 2 (0x00000002)

WIPE_SILENTLY

```
public static final int WIPE_SILENTLY
```

Flag for `wipeData(int)` : won't show reason for wiping to the user.

Constant Value: 8 (0x00000008)

Public methods

acknowledgeDeviceCompliant

```
public void acknowledgeDeviceCompliant ()
```

Called by a profile owner of an organization-owned managed profile to acknowledge that the device is compliant and the user can turn the profile off if needed according to the maximum time off policy. This method should be called when the device is deemed compliant after getting `DeviceAdminReceiver.onComplianceAcknowledgementRequired(Context,Intent)` callback in case it is overridden. Before this method is called the user is still free to turn the profile off, but the timer won't be reset, so personal apps will be suspended sooner. DPCs only need acknowledging device compliance if they override `DeviceAdminReceiver.onComplianceAcknowledgementRequired(Context,Intent)` , otherwise compliance is acknowledged automatically.

Throws	
`IllegalStateException`	if the user isn't unlocked

See also:

- `isComplianceAcknowledgementRequired()`
- `setManagedProfileMaximumTimeOff(ComponentName,long)`
- `DeviceAdminReceiver.onComplianceAcknowledgementRequired(Context,Intent)`

addCrossProfileIntentFilter

```
public void addCrossProfileIntentFilter (ComponentName admin,
                IntentFilter filter,
```

```
                int flags)
```

Called by the profile owner of a managed profile so that some intents sent in the managed profile can also be resolved in the parent, or vice versa. Only activity intents are supported.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value may be `null` .
`filter`	`IntentFilter` : The `IntentFilter` the intent has to match to be also resolved in the other profile
`flags`	`int` : `DevicePolicyManager.FLAG_MANAGED_CAN_ACCESS_PARENT` and `DevicePolicyManager.FLAG_PARENT_CAN_ACCESS_MANAGED` are supported.

Throws	
`SecurityException`	if `admin` is not a device or profile owner.

addCrossProfileWidgetProvider

```
public boolean addCrossProfileWidgetProvider (ComponentName admin,
                String packageName)
```

Called by the profile owner of a managed profile or a holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_PROFILE_INTERACTION` to enable widget providers from a given package to be available in the parent profile. As a result the user will be able to add widgets from the allowlisted package running under the profile to a widget host which runs under the parent profile, for example the home screen. Note that a package may have zero or more provider components, where each component provides a different widget type.

Note: By default no widget provider package is allowlisted. This API updates the allowlist incrementally. For better performance when updating multiple providers, use `setCrossProfileWidgetProviders(Set)` .

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null` .
`packageName`	`String` : The package from which widget providers are allowlisted.

Returns	
`boolean`	Whether the package was added.

Throws

`SecurityException`	if `admin` is not a profile owner and not a holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_PROFILE_INTERACTION` .

See also:

- `removeCrossProfileWidgetProvider(android.content.ComponentName,String)`
- `getCrossProfileWidgetProviders(android.content.ComponentName)`

addOverrideApn

```
public int addOverrideApn (ComponentName admin,
                ApnSetting apnSetting)
```

Called by device owner or managed profile owner to add an override APN.

This method may returns `-1` if `apnSetting` conflicts with an existing override APN. Update the existing conflicted APN with `updateOverrideApn(ComponentName,int,ApnSetting)` instead of adding a new entry.

Two override APNs are considered to conflict when all the following APIs return the same values on both override APNs:

- `ApnSetting.getOperatorNumeric()`
- `ApnSetting.getApnName()`
- `ApnSetting.getProxyAddressAsString()`
- `ApnSetting.getProxyPort()`
- `ApnSetting.getMmsProxyAddressAsString()`
- `ApnSetting.getMmsProxyPort()`
- `ApnSetting.getMmsc()`
- `ApnSetting.isEnabled()`
- `ApnSetting.getMvnoType()`
- `ApnSetting.getProtocol()`
- `ApnSetting.getRoamingProtocol()`

Before Android version `Build.VERSION_CODES.TIRAMISU` : Only device owners can add APNs.

Starting from Android version `Build.VERSION_CODES.TIRAMISU` : Both device owners and managed profile owners can add enterprise APNs (`ApnSetting.TYPE_ENTERPRISE`), while only device owners can add other type of APNs. Enterprise APNs are specific to the managed profile and do not override any user-configured VPNs. They are prerequisites for enabling preferential network service on the managed profile on 4G networks (`setPreferentialNetworkServiceConfigs(List)`).

Parameters	
`admin`	`ComponentName` : which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .

`apnSetting`	`ApnSetting` : the override APN to insert. This value cannot be `null` .
Returns	
`int`	The `id` of inserted override APN. Or `-1` when failed to insert into the database.
Throws	
`Security Exception`	If request is for enterprise APN `admin` is either device owner or profile owner and in all other types of APN if `admin` is not a device owner.

addPersistentPreferredActivity

```
public void addPersistentPreferredActivity (ComponentName admin,
                IntentFilter filter,
                ComponentName activity)
```

Called by a profile owner or device owner or holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_LOCK_TASK` . to set a default activity that the system selects to handle intents that match the given `IntentFilter` instead of showing the default disambiguation mechanism. This activity will remain the default intent handler even if the set of potential event handlers for the intent filter changes and if the intent preferences are reset.

Note that the target application should still declare the activity in the manifest, the API just sets the activity to be the default one to handle the given intent filter.

The default disambiguation mechanism takes over if the activity is not installed (anymore). When the activity is (re)installed, it is automatically reset as default intent handler for the filter.

Note that calling this API to set a default intent handler, only allow to avoid the default disambiguation mechanism. Implicit intents that do not trigger this mechanism (like invoking the browser) cannot be configured as they are controlled by other configurations.

The calling device admin must be a profile owner or device owner. If it is not, a security exception will be thrown.

Starting from `Build.VERSION_CODES.UPSIDE_DOWN_CAKE` , after the persistent preferred activity policy has been set, `PolicyUpdateReceiver.onPolicySetResult(Context,String,Bundle,TargetUser,PolicyUpdateResult)` will notify the admin on whether the policy was successfully set or not. This callback will contain:

- The policy identifier `DevicePolicyIdentifiers.PERSISTENT_PREFERRED_ACTIVITY_POLICY`
- The additional policy params bundle, which contains `PolicyUpdateReceiver.EXTRA_INTENT_FILTER` the intent filter the policy applies to
- The `TargetUser` that this policy relates to
- The `PolicyUpdateResult` , which will be `PolicyUpdateResult.RESULT_POLICY_SET` if the policy was successfully set or the reason the policy failed to be set (e.g. `PolicyUpdateResult.RESULT_FAILURE_CONFLICTING_ADMIN_POLICY`)

If there has been a change to the policy, `PolicyUpdateReceiver.onPolicyChanged(Context,String,Bundle,TargetUser,PolicyUpdateResult)` will notify the admin of this change. This callback will contain the same parameters as PolicyUpdateReceiver#onPolicySetResult and the `PolicyUpdateResult` will contain the reason why the policy changed.

NOTE: Performs disk I/O and shouldn't be called on the main thread.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null` .
`filter`	`IntentFilter` : The IntentFilter for which a default handler is added.
`activity`	`ComponentName` : The Activity that is added as default intent handler. This value cannot be `null` .

Throws	
`SecurityException`	if `admin` is not a device or profile owner or holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_LOCK_TASK` .

addUserRestriction

```
public void addUserRestriction (ComponentName admin,
                String key)
```

Called by a profile owner, device owner or a holder of any permission that is associated with a user restriction to set a user restriction specified by the key.

The calling device admin must be a profile owner, device owner or holder of any permission that is associated with a user restriction; if it is not, a security exception will be thrown.

The profile owner of an organization-owned managed profile may invoke this method on the `DevicePolicyManager` instance it obtained from `getParentProfileInstance(ComponentName)` , for enforcing device-wide restrictions.

See the constants in `UserManager` for the list of restrictions that can be enforced device-wide. These constants will also state in their documentation which permission is required to manage the restriction using this API.

Starting from `Build.VERSION_CODES.UPSIDE_DOWN_CAKE` , after the user restriction policy has been set, `PolicyUpdateReceiver.onPolicySetResult(Context,String,Bundle,TargetUser,PolicyUpdateResult)` will notify the admin on whether the policy was successfully set or not. This callback will contain:

- The policy identifier returned from `DevicePolicyIdentifiers.getIdentifierForUserRestriction(String)`
- The `TargetUser` that this policy relates to
- The `PolicyUpdateResult` , which will be `PolicyUpdateResult.RESULT_POLICY_SET` if the policy was successfully set or the reason the policy failed to be set (e.g.

`PolicyUpdateResult.RESULT_FAILURE_CONFLICTING_ADMIN_POLICY`)

If there has been a change to the policy, `PolicyUpdateReceiver.onPolicyChanged(Context,String,Bundle,TargetUser,PolicyUpdateResult)` will notify the admin of this change. This callback will contain the same parameters as PolicyUpdateReceiver#onPolicySetResult and the `PolicyUpdateResult` will contain the reason why the policy changed.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with or `null` if the caller is not a device admin.
`key`	`String` : The key of the restriction. Value is one of the following: • `UserManager.ALLOW_PARENT_PROFILE_APP_LINKING` • `UserManager.DISALLOW_ADD_MANAGED_PROFILE` • `UserManager.DISALLOW_ADD_PRIVATE_PROFILE` • `UserManager.DISALLOW_ADD_USER` • `UserManager.DISALLOW_ADD_WIFI_CONFIG` • `UserManager.DISALLOW_ADJUST_VOLUME` • `UserManager.DISALLOW_AIRPLANE_MODE` • `UserManager.DISALLOW_AMBIENT_DISPLAY` • `UserManager.DISALLOW_APPS_CONTROL` • `UserManager.DISALLOW_ASSIST_CONTENT` • `UserManager.DISALLOW_AUTOFILL` • `UserManager.DISALLOW_BLUETOOTH` • `UserManager.DISALLOW_BLUETOOTH_SHARING` • `UserManager.DISALLOW_CAMERA_TOGGLE` • `UserManager.DISALLOW_CELLULAR_2G` • `UserManager.DISALLOW_CHANGE_NEAR_FIELD_COMMUNICATION_RADIO` • `UserManager.DISALLOW_CHANGE_WIFI_STATE` • `UserManager.DISALLOW_CONFIG_BLUETOOTH` • `UserManager.DISALLOW_CONFIG_BRIGHTNESS` • `UserManager.DISALLOW_CONFIG_CELL_BROADCASTS` • `UserManager.DISALLOW_CONFIG_CREDENTIALS` • `UserManager.DISALLOW_CONFIG_DATE_TIME` • `UserManager.DISALLOW_CONFIG_DEFAULT_APPS` • `UserManager.DISALLOW_CONFIG_LOCALE` • `UserManager.DISALLOW_CONFIG_LOCATION` • `UserManager.DISALLOW_CONFIG_MOBILE_NETWORKS` • `UserManager.DISALLOW_CONFIG_PRIVATE_DNS` • `UserManager.DISALLOW_CONFIG_SCREEN_TIMEOUT` • `UserManager.DISALLOW_CONFIG_TETHERING` • `UserManager.DISALLOW_CONFIG_VPN` • `UserManager.DISALLOW_CONFIG_WIFI`

- `UserManager.DISALLOW_CONTENT_CAPTURE`
- `UserManager.DISALLOW_CONTENT_SUGGESTIONS`
- `UserManager.DISALLOW_CREATE_WINDOWS`
- `UserManager.DISALLOW_CROSS_PROFILE_COPY_PASTE`
- `UserManager.DISALLOW_DATA_ROAMING`
- `UserManager.DISALLOW_DEBUGGING_FEATURES`
- `UserManager.DISALLOW_FACTORY_RESET`
- `UserManager.DISALLOW_FUN`
- `UserManager.DISALLOW_GRANT_ADMIN`
- `UserManager.DISALLOW_TASK_CONTINUITY_HANDOFF`
- `UserManager.DISALLOW_INSTALL_APPS`
- `UserManager.DISALLOW_INSTALL_UNKNOWN_SOURCES`
- `UserManager.DISALLOW_INSTALL_UNKNOWN_SOURCES_GLOBALLY`
- `UserManager.DISALLOW_MICROPHONE_TOGGLE`
- `UserManager.DISALLOW_MODIFY_ACCOUNTS`
- `UserManager.DISALLOW_MOUNT_PHYSICAL_MEDIA`
- `UserManager.DISALLOW_NEAR_FIELD_COMMUNICATION_RADIO`
- `UserManager.DISALLOW_NETWORK_RESET`
- `UserManager.DISALLOW_OUTGOING_BEAM`
- `UserManager.DISALLOW_OUTGOING_CALLS`
- `UserManager.DISALLOW_PRINTING`
- `UserManager.DISALLOW_REMOVE_MANAGED_PROFILE`
- `UserManager.DISALLOW_REMOVE_USER`
- `UserManager.DISALLOW_SAFE_BOOT`
- `UserManager.DISALLOW_SET_USER_ICON`
- `UserManager.DISALLOW_SET_WALLPAPER`
- `UserManager.DISALLOW_SHARE_INTO_MANAGED_PROFILE`
- `UserManager.DISALLOW_SHARE_LOCATION`
- `UserManager.DISALLOW_SHARING_ADMIN_CONFIGURED_WIFI`
- `UserManager.DISALLOW_SIM_GLOBALLY`
- `UserManager.DISALLOW_SMS`
- `UserManager.DISALLOW_SYSTEM_ERROR_DIALOGS`
- `UserManager.DISALLOW_THREAD_NETWORK`
- `UserManager.DISALLOW_ULTRA_WIDEBAND_RADIO`
- `UserManager.DISALLOW_UNIFIED_PASSWORD`
- `UserManager.DISALLOW_UNINSTALL_APPS`
- `UserManager.DISALLOW_UNMUTE_MICROPHONE`
- `UserManager.DISALLOW_USB_FILE_TRANSFER`
- `UserManager.DISALLOW_USER_SWITCH`
- `UserManager.DISALLOW_WIFI_DIRECT`
- `UserManager.DISALLOW_WIFI_TETHERING`
- `UserManager.ENSURE_VERIFY_APPS`
- `UserManager.KEY_RESTRICTIONS_PENDING`

Throws	
`Security Exception`	if `admin` is not a device or profile owner and if the caller has not been granted the permission to set the given user restriction.

addUserRestrictionGlobally

```
public void addUserRestrictionGlobally (String key)
```

Called by a profile owner, device owner or a holder of any permission that is associated with a user restriction to set a user restriction specified by the provided `key` globally on all users. To clear the restriction use `clearUserRestriction(ComponentName, String)`.

For a given user, a restriction will be set if it was applied globally or locally by any admin.

The calling device admin must be a profile owner, device owner or a holder of any permission that is associated with a user restriction; if it is not, a security exception will be thrown.

See the constants in `UserManager` for the list of restrictions that can be enforced device-wide. These constants will also state in their documentation which permission is required to manage the restriction using this API.

After the user restriction policy has been set, `PolicyUpdateReceiver.onPolicySetResult(Context,String,Bundle,TargetUser,PolicyUpdateResult)` will notify the admin on whether the policy was successfully set or not. This callback will contain:

- The policy identifier returned from `DevicePolicyIdentifiers.getIdentifierForUserRestriction(String)`
- The `TargetUser` that this policy relates to
- The `PolicyUpdateResult`, which will be `PolicyUpdateResult.RESULT_POLICY_SET` if the policy was successfully set or the reason the policy failed to be set (e.g. `PolicyUpdateResult.RESULT_FAILURE_CONFLICTING_ADMIN_POLICY`)

If there has been a change to the policy, `PolicyUpdateReceiver.onPolicyChanged(Context,String,Bundle,TargetUser,PolicyUpdateResult)` will notify the admin of this change. This callback will contain the same parameters as PolicyUpdateReceiver#onPolicySetResult and the `PolicyUpdateResult` will contain the reason why the policy changed.

Parameters	
`key`	`String`: The key of the restriction. This value cannot be `null`. Value is one of the following: • `UserManager.ALLOW_PARENT_PROFILE_APP_LINKING` • `UserManager.DISALLOW_ADD_MANAGED_PROFILE` • `UserManager.DISALLOW_ADD_PRIVATE_PROFILE` • `UserManager.DISALLOW_ADD_USER` • `UserManager.DISALLOW_ADD_WIFI_CONFIG`

- `UserManager.DISALLOW_ADJUST_VOLUME`
- `UserManager.DISALLOW_AIRPLANE_MODE`
- `UserManager.DISALLOW_AMBIENT_DISPLAY`
- `UserManager.DISALLOW_APPS_CONTROL`
- `UserManager.DISALLOW_ASSIST_CONTENT`
- `UserManager.DISALLOW_AUTOFILL`
- `UserManager.DISALLOW_BLUETOOTH`
- `UserManager.DISALLOW_BLUETOOTH_SHARING`
- `UserManager.DISALLOW_CAMERA_TOGGLE`
- `UserManager.DISALLOW_CELLULAR_2G`
- `UserManager.DISALLOW_CHANGE_NEAR_FIELD_COMMUNICATION_RADIO`
- `UserManager.DISALLOW_CHANGE_WIFI_STATE`
- `UserManager.DISALLOW_CONFIG_BLUETOOTH`
- `UserManager.DISALLOW_CONFIG_BRIGHTNESS`
- `UserManager.DISALLOW_CONFIG_CELL_BROADCASTS`
- `UserManager.DISALLOW_CONFIG_CREDENTIALS`
- `UserManager.DISALLOW_CONFIG_DATE_TIME`
- `UserManager.DISALLOW_CONFIG_DEFAULT_APPS`
- `UserManager.DISALLOW_CONFIG_LOCALE`
- `UserManager.DISALLOW_CONFIG_LOCATION`
- `UserManager.DISALLOW_CONFIG_MOBILE_NETWORKS`
- `UserManager.DISALLOW_CONFIG_PRIVATE_DNS`
- `UserManager.DISALLOW_CONFIG_SCREEN_TIMEOUT`
- `UserManager.DISALLOW_CONFIG_TETHERING`
- `UserManager.DISALLOW_CONFIG_VPN`
- `UserManager.DISALLOW_CONFIG_WIFI`
- `UserManager.DISALLOW_CONTENT_CAPTURE`
- `UserManager.DISALLOW_CONTENT_SUGGESTIONS`
- `UserManager.DISALLOW_CREATE_WINDOWS`
- `UserManager.DISALLOW_CROSS_PROFILE_COPY_PASTE`
- `UserManager.DISALLOW_DATA_ROAMING`
- `UserManager.DISALLOW_DEBUGGING_FEATURES`
- `UserManager.DISALLOW_FACTORY_RESET`
- `UserManager.DISALLOW_FUN`
- `UserManager.DISALLOW_GRANT_ADMIN`
- `UserManager.DISALLOW_TASK_CONTINUITY_HANDOFF`
- `UserManager.DISALLOW_INSTALL_APPS`
- `UserManager.DISALLOW_INSTALL_UNKNOWN_SOURCES`
- `UserManager.DISALLOW_INSTALL_UNKNOWN_SOURCES_GLOBALLY`
- `UserManager.DISALLOW_MICROPHONE_TOGGLE`
- `UserManager.DISALLOW_MODIFY_ACCOUNTS`
- `UserManager.DISALLOW_MOUNT_PHYSICAL_MEDIA`
- `UserManager.DISALLOW_NEAR_FIELD_COMMUNICATION_RADIO`
- `UserManager.DISALLOW_NETWORK_RESET`

	• UserManager.DISALLOW_OUTGOING_BEAM • UserManager.DISALLOW_OUTGOING_CALLS • UserManager.DISALLOW_PRINTING • UserManager.DISALLOW_REMOVE_MANAGED_PROFILE • UserManager.DISALLOW_REMOVE_USER • UserManager.DISALLOW_SAFE_BOOT • UserManager.DISALLOW_SET_USER_ICON • UserManager.DISALLOW_SET_WALLPAPER • UserManager.DISALLOW_SHARE_INTO_MANAGED_PROFILE • UserManager.DISALLOW_SHARE_LOCATION • UserManager.DISALLOW_SHARING_ADMIN_CONFIGURED_WIFI • UserManager.DISALLOW_SIM_GLOBALLY • UserManager.DISALLOW_SMS • UserManager.DISALLOW_SYSTEM_ERROR_DIALOGS • UserManager.DISALLOW_THREAD_NETWORK • UserManager.DISALLOW_ULTRA_WIDEBAND_RADIO • UserManager.DISALLOW_UNIFIED_PASSWORD • UserManager.DISALLOW_UNINSTALL_APPS • UserManager.DISALLOW_UNMUTE_MICROPHONE • UserManager.DISALLOW_USB_FILE_TRANSFER • UserManager.DISALLOW_USER_SWITCH • UserManager.DISALLOW_WIFI_DIRECT • UserManager.DISALLOW_WIFI_TETHERING • UserManager.ENSURE_VERIFY_APPS • UserManager.KEY_RESTRICTIONS_PENDING

Throws	
IllegalStateException	if caller is not targeting Android Build.VERSION_CODES.UPSIDE_DOWN_CAKE or above.
SecurityException	if `admin` is not a device or profile owner and if the caller has not been granted the permission to set the given user restriction.

bindDeviceAdminServiceAsUser

```
public boolean bindDeviceAdminServiceAsUser (ComponentName admin,
                Intent serviceIntent,
                ServiceConnection conn,
                int flags,
                UserHandle targetUser)
```

Called by a device owner to bind to a service from a secondary managed user or vice versa. See `getBindDeviceAdminTargetUsers(ComponentName)` for the pre-requirements of a device owner to bind to services of another managed user.

The service must be protected by `Manifest.permission.BIND_DEVICE_ADMIN`. Note that the `Context` used to obtain this `DevicePolicyManager` instance via `Context.getSystemService(Class)` will be used to bind to the `Service`.

Note: This method used to be available for communication between device owner and profile owner. However, since Android 11, this combination is not possible. This method is now only useful for communication between device owner and managed secondary users.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null`.
`service Intent`	`Intent`: Identifies the service to connect to. The Intent must specify either an explicit component name or a package name to match an `IntentFilter` published by a service. This value cannot be `null`.
`conn`	`ServiceConnection`: Receives information as the service is started and stopped in main thread. This must be a valid `ServiceConnection` object; it must not be `null`.
`flags`	`int`: Operation options for the binding operation. See `Context.bindService(Intent,ServiceConnection,int)`. Value is either `0` or a combination of the following: • `Context.BIND_AUTO_CREATE` • `Context.BIND_DEBUG_UNBIND` • `Context.BIND_NOT_FOREGROUND` • `Context.BIND_ABOVE_CLIENT` • `Context.BIND_ALLOW_OOM_MANAGEMENT` • `Context.BIND_WAIVE_PRIORITY` • `Context.BIND_IMPORTANT` • `Context.BIND_ADJUST_WITH_ACTIVITY` • `Context.BIND_NOT_PERCEPTIBLE` • `Context.BIND_ALLOW_ACTIVITY_STARTS` • `Context.BIND_INCLUDE_CAPABILITIES` • `Context.BIND_SHARED_ISOLATED_PROCESS` • `Context.BIND_PACKAGE_ISOLATED_PROCESS` • `Context.BIND_EXTERNAL_SERVICE`
`target User`	`UserHandle`: Which user to bind to. Must be one of the users returned by `getBindDeviceAdminTargetUsers(ComponentName)`, otherwise a `SecurityException` will be thrown. This value cannot be `null`.
Returns	
`boolean`	If you have successfully bound to the service, `true` is returned; `false` is returned if the connection is not made and you will not receive the service object.

See also:

- `Context.bindService(Intent,ServiceConnection,int)`
- `getBindDeviceAdminTargetUsers(ComponentName)`

bindDeviceAdminServiceAsUser

```
public boolean bindDeviceAdminServiceAsUser (ComponentName admin,
                Intent serviceIntent,
                ServiceConnection conn,
                Context.BindServiceFlags flags,
                UserHandle targetUser)
```

See `bindDeviceAdminServiceAsUser(ComponentName,Intent,ServiceConnection,int,UserHandle)` . Call `Context.BindServiceFlags.of(long)` to obtain a BindServiceFlags object.

Parameters	
`admin`	`ComponentName` : This value cannot be `null` .
`serviceIntent`	`Intent` : This value cannot be `null` .
`conn`	`ServiceConnection` : This value cannot be `null` .
`flags`	`Context.BindServiceFlags` : This value cannot be `null` .
`targetUser`	`UserHandle` : This value cannot be `null` .

Returns	
`boolean`	

canAdminGrantSensorsPermissions

```
public boolean canAdminGrantSensorsPermissions ()
```

Returns true if the caller is running on a device where an admin can grant permissions related to device sensors. This is a signal that the device is a fully-managed device where personal usage is discouraged. The list of permissions is listed in `setPermissionGrantState(ComponentName,String,String,int)` . May be called by any app.

Returns	
`boolean`	true if an admin can grant device sensors-related permissions, false otherwise.

canUsbDataSignalingBeDisabled

```
public boolean canUsbDataSignalingBeDisabled ()
```

Returns whether enabling or disabling USB data signaling is supported on the device.

Returns	
`boolean`	`true` if the device supports enabling and disabling USB data signaling.

clearApplicationUserData

```
public void clearApplicationUserData (ComponentName admin,
                String packageName,
                Executor executor,
                DevicePolicyManager.OnClearApplicationUserDataListener listener)
```

Called by the device owner or profile owner to clear application user data of a given package. The behaviour of this is equivalent to the target application calling `ActivityManager.clearApplicationUserData()`.

Note: an application can store data outside of its application data, e.g. external storage or user dictionary. This data will not be wiped by calling this API.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null`.
`packageName`	`String`: The name of the package which will have its user data wiped. This value cannot be `null`.
`executor`	`Executor`: The executor through which the listener should be invoked. This value cannot be `null`. Callback and listener events are dispatched through this `Executor`, providing an easy way to control which thread is used. To dispatch events through the main thread of your application, you can use `Context.getMainExecutor()`. Otherwise, provide an `Executor` that dispatches to an appropriate thread.
`listener`	`DevicePolicyManager.OnClearApplicationUserDataListener`: A callback object that will inform the caller when the clearing is done. This value cannot be `null`.

Throws	
`SecurityException`	if the caller is not the device owner/profile owner.

clearCrossProfileIntentFilters

```
public void clearCrossProfileIntentFilters (ComponentName admin)
```

Called by a profile owner of a managed profile to remove the cross-profile intent filters that go from the managed profile to the parent, or from the parent to the managed profile. Only removes those that have been set by the profile owner.

Note: A list of default cross profile intent filters are set up by the system when the profile is created, some of them ensure the proper functioning of the profile, while others enable sharing of data from the parent to the managed profile for user convenience. These default intent filters are not cleared when this API is called. If the default cross profile data sharing is not desired, they can be disabled with `UserManager.DISALLOW_SHARE_INTO_MANAGED_PROFILE`.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. This value may be `null`.
Throws	
`SecurityException`	if `admin` is not a profile owner.

clearDeviceOwnerApp

```
public void clearDeviceOwnerApp (String packageName)
```

This method was deprecated in API level 26.
This method is expected to be used for testing purposes only. The device owner will lose control of the device and its data after calling it. In order to protect any sensitive data that remains on the device, it is advised that the device owner factory resets the device instead of calling this method. See `wipeData(int)`.

Clears the current device owner. The caller must be the device owner. This function should be used cautiously as once it is called it cannot be undone. The device owner can only be set as a part of device setup, before it completes.

While some policies previously set by the device owner will be cleared by this method, it is a best-effort process and some other policies will still remain in place after the device owner is cleared.

Parameters	
`packageName`	`String`: The package name of the device owner.
Throws	
`SecurityException`	if the caller is not in `packageName` or `packageName` does not own the current device owner component.

clearPackagePersistentPreferredActivities

```
public void clearPackagePersistentPreferredActivities (ComponentName admin,
                String packageName)
```

Called by a profile owner or device owner or holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_LOCK_TASK` to remove all persistent intent handler preferences associated with the given package that were set by `addPersistentPreferredActivity(ComponentName, IntentFilter, ComponentName)`.

The calling device admin must be a profile owner. If it is not, a security exception will be thrown.

Starting from `Build.VERSION_CODES.UPSIDE_DOWN_CAKE`, after the persistent preferred activity policy has been cleared, `PolicyUpdateReceiver.onPolicySetResult(Context,String,Bundle,TargetUser,PolicyUpdateResult)` will notify the admin on whether the policy was successfully cleared or not. This callback will contain:

- The policy identifier `DevicePolicyIdentifiers.PERSISTENT_PREFERRED_ACTIVITY_POLICY`
- The additional policy params bundle, which contains `PolicyUpdateReceiver.EXTRA_INTENT_FILTER` the intent filter the policy applies to
- The `TargetUser` that this policy relates to
- The `PolicyUpdateResult`, which will be `PolicyUpdateResult.RESULT_POLICY_SET` if the policy was successfully cleared or the reason the policy failed to be cleared (e.g. `PolicyUpdateResult.RESULT_FAILURE_CONFLICTING_ADMIN_POLICY`)

If there has been a change to the policy, `PolicyUpdateReceiver.onPolicyChanged(Context,String,Bundle,TargetUser,PolicyUpdateResult)` will notify the admin of this change. This callback will contain the same parameters as PolicyUpdateReceiver#onPolicySetResult and the `PolicyUpdateResult` will contain the reason why the policy changed.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null`.
`packageName`	`String`: The name of the package for which preferences are removed.
Throws	
`SecurityException`	if `admin` is not a device or profile owner or holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_LOCK_TASK`.

clearProfileOwner

```
public void clearProfileOwner (ComponentName admin)
```

This method was deprecated in API level 26.
This method is expected to be used for testing purposes only. The profile owner will lose control of the user and its data after calling it. In order to protect any sensitive data that remains on this user, it is advised that the profile owner deletes it instead of calling this method. See `wipeData(int)`.

Clears the active profile owner. The caller must be the profile owner of this user, otherwise a SecurityException will be thrown. This method is not available to managed profile owners.

While some policies previously set by the profile owner will be cleared by this method, it is a best-effort process and some other policies will still remain in place after the profile owner is cleared.

Parameters	
`admin`	`ComponentName` : The component to remove as the profile owner. This value cannot be `null` .
Throws	
`SecurityException`	if `admin` is not an active profile owner, or the method is being called from a managed profile.

clearResetPasswordToken

```
public boolean clearResetPasswordToken (ComponentName admin)
```

Called by a profile, device owner or holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_RESET_PASSWORD` to revoke the current password reset token.

On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, this method has no effect - the reset token should not have been set in the first place - and false is returned.
Requires the `PackageManager#FEATURE_SECURE_LOCK_SCREEN` feature which can be detected using `PackageManager.hasSystemFeature(String)` .

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null` .
Returns	
`boolean`	true if the operation is successful, false otherwise.
Throws	
`SecurityException`	if admin is not a device or profile owner and if the caller does not the permission `Manifest.permission.MANAGE_DEVICE_POLICY_RESET_PASSWORD` .

clearUserRestriction

```
public void clearUserRestriction (ComponentName admin,
                String key)
```

Called by a profile owner, device owner or a holder of any permission that is associated with a user restriction to clear a user restriction specified by the key.

The calling device admin must be a profile or device owner; if it is not, a security exception will be thrown.

The profile owner of an organization-owned managed profile may invoke this method on the `DevicePolicyManager` instance it obtained from `getParentProfileInstance(ComponentName)`, for clearing device-wide restrictions.

See the constants in `UserManager` for the list of restrictions. These constants state in their documentation which permission is required to manage the restriction using this API.

Starting from `Build.VERSION_CODES.UPSIDE_DOWN_CAKE`, after the user restriction policy has been cleared, `PolicyUpdateReceiver.onPolicySetResult(Context,String,Bundle,TargetUser,PolicyUpdateResult)` will notify the admin on whether the policy was successfully cleared or not. This callback will contain:

- The policy identifier returned from `DevicePolicyIdentifiers.getIdentifierForUserRestriction(String)`
- The `TargetUser` that this policy relates to
- The `PolicyUpdateResult`, which will be `PolicyUpdateResult.RESULT_POLICY_SET` if the policy was successfully cleared or the reason the policy failed to be cleared (e.g. `PolicyUpdateResult.RESULT_FAILURE_CONFLICTING_ADMIN_POLICY`)

If there has been a change to the policy, `PolicyUpdateReceiver.onPolicyChanged(Context,String,Bundle,TargetUser,PolicyUpdateResult)` will notify the admin of this change. This callback will contain the same parameters as PolicyUpdateReceiver#onPolicySetResult and the `PolicyUpdateResult` will contain the reason why the policy changed.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with or `null` if the caller is not a device admin.
`key`	`String`: The key of the restriction. Value is one of the following: • `UserManager.ALLOW_PARENT_PROFILE_APP_LINKING` • `UserManager.DISALLOW_ADD_MANAGED_PROFILE` • `UserManager.DISALLOW_ADD_PRIVATE_PROFILE` • `UserManager.DISALLOW_ADD_USER` • `UserManager.DISALLOW_ADD_WIFI_CONFIG` • `UserManager.DISALLOW_ADJUST_VOLUME` • `UserManager.DISALLOW_AIRPLANE_MODE` • `UserManager.DISALLOW_AMBIENT_DISPLAY` • `UserManager.DISALLOW_APPS_CONTROL` • `UserManager.DISALLOW_ASSIST_CONTENT` • `UserManager.DISALLOW_AUTOFILL` • `UserManager.DISALLOW_BLUETOOTH` • `UserManager.DISALLOW_BLUETOOTH_SHARING` • `UserManager.DISALLOW_CAMERA_TOGGLE`

- `UserManager.DISALLOW_CELLULAR_2G`
- `UserManager.DISALLOW_CHANGE_NEAR_FIELD_COMMUNICATION_RADIO`
- `UserManager.DISALLOW_CHANGE_WIFI_STATE`
- `UserManager.DISALLOW_CONFIG_BLUETOOTH`
- `UserManager.DISALLOW_CONFIG_BRIGHTNESS`
- `UserManager.DISALLOW_CONFIG_CELL_BROADCASTS`
- `UserManager.DISALLOW_CONFIG_CREDENTIALS`
- `UserManager.DISALLOW_CONFIG_DATE_TIME`
- `UserManager.DISALLOW_CONFIG_DEFAULT_APPS`
- `UserManager.DISALLOW_CONFIG_LOCALE`
- `UserManager.DISALLOW_CONFIG_LOCATION`
- `UserManager.DISALLOW_CONFIG_MOBILE_NETWORKS`
- `UserManager.DISALLOW_CONFIG_PRIVATE_DNS`
- `UserManager.DISALLOW_CONFIG_SCREEN_TIMEOUT`
- `UserManager.DISALLOW_CONFIG_TETHERING`
- `UserManager.DISALLOW_CONFIG_VPN`
- `UserManager.DISALLOW_CONFIG_WIFI`
- `UserManager.DISALLOW_CONTENT_CAPTURE`
- `UserManager.DISALLOW_CONTENT_SUGGESTIONS`
- `UserManager.DISALLOW_CREATE_WINDOWS`
- `UserManager.DISALLOW_CROSS_PROFILE_COPY_PASTE`
- `UserManager.DISALLOW_DATA_ROAMING`
- `UserManager.DISALLOW_DEBUGGING_FEATURES`
- `UserManager.DISALLOW_FACTORY_RESET`
- `UserManager.DISALLOW_FUN`
- `UserManager.DISALLOW_GRANT_ADMIN`
- `UserManager.DISALLOW_TASK_CONTINUITY_HANDOFF`
- `UserManager.DISALLOW_INSTALL_APPS`
- `UserManager.DISALLOW_INSTALL_UNKNOWN_SOURCES`
- `UserManager.DISALLOW_INSTALL_UNKNOWN_SOURCES_GLOBALLY`
- `UserManager.DISALLOW_MICROPHONE_TOGGLE`
- `UserManager.DISALLOW_MODIFY_ACCOUNTS`
- `UserManager.DISALLOW_MOUNT_PHYSICAL_MEDIA`
- `UserManager.DISALLOW_NEAR_FIELD_COMMUNICATION_RADIO`
- `UserManager.DISALLOW_NETWORK_RESET`
- `UserManager.DISALLOW_OUTGOING_BEAM`
- `UserManager.DISALLOW_OUTGOING_CALLS`
- `UserManager.DISALLOW_PRINTING`
- `UserManager.DISALLOW_REMOVE_MANAGED_PROFILE`
- `UserManager.DISALLOW_REMOVE_USER`
- `UserManager.DISALLOW_SAFE_BOOT`
- `UserManager.DISALLOW_SET_USER_ICON`
- `UserManager.DISALLOW_SET_WALLPAPER`
- `UserManager.DISALLOW_SHARE_INTO_MANAGED_PROFILE`

	• UserManager.DISALLOW_SHARE_LOCATION • UserManager.DISALLOW_SHARING_ADMIN_CONFIGURED_WIFI • UserManager.DISALLOW_SIM_GLOBALLY • UserManager.DISALLOW_SMS • UserManager.DISALLOW_SYSTEM_ERROR_DIALOGS • UserManager.DISALLOW_THREAD_NETWORK • UserManager.DISALLOW_ULTRA_WIDEBAND_RADIO • UserManager.DISALLOW_UNIFIED_PASSWORD • UserManager.DISALLOW_UNINSTALL_APPS • UserManager.DISALLOW_UNMUTE_MICROPHONE • UserManager.DISALLOW_USB_FILE_TRANSFER • UserManager.DISALLOW_USER_SWITCH • UserManager.DISALLOW_WIFI_DIRECT • UserManager.DISALLOW_WIFI_TETHERING • UserManager.ENSURE_VERIFY_APPS • UserManager.KEY_RESTRICTIONS_PENDING

Throws	
SecurityException	if `admin` is not a device or profile owner and if the caller has not been granted the permission to set the given user restriction.

createAdminSupportIntent

```
public Intent createAdminSupportIntent (String restriction)
```

Called by any app to display a support dialog when a feature was disabled by an admin. This returns an intent that can be used with `Context.startActivity(Intent)` to display the dialog. It will tell the user that the feature indicated by `restriction` was disabled by an admin, and include a link for more information. The default content of the dialog can be changed by the restricting admin via `setShortSupportMessage(ComponentName,CharSequence)`. If the restriction is not set (i.e. the feature is available), then the return value will be `null`.

Parameters	
`restriction`	`String`: Indicates for which feature the dialog should be displayed. Can be a user restriction from `UserManager`, e.g. `UserManager.DISALLOW_ADJUST_VOLUME`, or one of the constants `POLICY_DISABLE_CAMERA` or `POLICY_DISABLE_SCREEN_CAPTURE`. This value cannot be `null`.

Returns	
Intent	Intent An intent to be used to start the dialog-activity if the restriction is set by an admin, or null if the restriction does not exist or no admin set it.

createAndManageUser

```
public UserHandle createAndManageUser (ComponentName admin,
                String name,
                ComponentName profileOwner,
                PersistableBundle adminExtras,
                int flags)
```

Called by a device owner to create a user with the specified name and a given component of the calling package as profile owner. The UserHandle returned by this method should not be persisted as user handles are recycled as users are removed and created. If you need to persist an identifier for this user, use `UserManager.getSerialNumberForUser` . The new user will not be started in the background.

admin is the `DeviceAdminReceiver` which is the device owner. profileOwner is also a DeviceAdminReceiver in the same package as admin, and will become the profile owner and will be registered as an active admin on the new user. The profile owner package will be installed on the new user.

If the adminExtras are not null, they will be stored on the device until the user is started for the first time. Then the extras will be passed to the admin when onEnable is called.

From `Build.VERSION_CODES.P` onwards, if targeting `Build.VERSION_CODES.P` , throws `UserOperationException` instead of returning `null` on failure.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
`name`	`String` : The user's name. This value cannot be `null` .
`profileOwner`	`ComponentName` : Which `DeviceAdminReceiver` will be profile owner. Has to be in the same package as admin, otherwise no user is created and an IllegalArgumentException is thrown. This value cannot be `null` .
`adminExtras`	`PersistableBundle` : Extras that will be passed to onEnable of the admin receiver on the new user. This value may be `null` .
`flags`	`int` : `SKIP_SETUP_WIZARD` , `MAKE_USER_EPHEMERAL` and `LEAVE_ALL_SYSTEM_APPS_ENABLED` are supported. Value is either `0` or a combination of the following: • `SKIP_SETUP_WIZARD` • `MAKE_USER_EPHEMERAL` • `LEAVE_ALL_SYSTEM_APPS_ENABLED`

Returns	
`UserHandle`	the `UserHandle` object for the created user, or `null` if the user could not be created.

Throws	
UserManager.UserOperationException	if the user could not be created and the calling app is targeting Build.VERSION_CODES.P and running on Build.VERSION_CODES.P .
SecurityException	if headless device is in DeviceAdminInfo.HEADLESS_DEVICE_OWNER_MODE_SINGLE_USER mode.
SecurityException	if admin is not a device owner

enableSystemApp

```
public int enableSystemApp (ComponentName admin,
                Intent intent)
```

Re-enable system apps by intent that were disabled by default when the user was initialized. This function can be called by a device owner, profile owner, or by a delegate given the DELEGATION_ENABLE_SYSTEM_APP scope via setDelegatedScopes(ComponentName, String, List) .

Parameters	
admin	ComponentName : Which DeviceAdminReceiver this request is associated with, or null if the caller is an enable system app delegate.
intent	Intent : An intent matching the app(s) to be installed. All apps that resolve for this intent will be re-enabled in the calling profile.

Returns	
int	int The number of activities that matched the intent and were installed.

Throws	
SecurityException	if admin is not a device or profile owner.

enableSystemApp

```
public void enableSystemApp (ComponentName admin,
                String packageName)
```

Re-enable a system app that was disabled by default when the user was initialized. This function can be called by a device owner, profile owner, or by a delegate given the DELEGATION_ENABLE_SYSTEM_APP scope via setDelegatedScopes(ComponentName, String, List) .

Parameters

`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with, or `null` if the caller is an enable system app delegate.
`packageName`	`String` : The package to be re-enabled in the calling profile.
Throws	
`SecurityException`	if `admin` is not a device or profile owner.

generateKeyPair

```
public AttestedKeyPair generateKeyPair (ComponentName admin,
                String algorithm,
                KeyGenParameterSpec keySpec,
                int idAttestationFlags)
```

This API can be called by the following to generate a new private/public key pair:

- Device owner
- Profile owner
- Delegated certificate installer
- Credential management app
- An app that holds the `Manifest.permission.MANAGE_DEVICE_POLICY_CERTIFICATES` permission

If the device supports key generation via secure hardware, this method is useful for creating a key in KeyChain that never left the secure hardware. Access to the key is controlled the same way as in `installKeyPair(ComponentName, PrivateKey, Certificate, String)` .

From Android `Build.VERSION_CODES.S` , the credential management app can call this API. If called by the credential management app, the componentName must be `null` . Note, there can only be a credential management app on an unmanaged device.

Because this method might take several seconds to complete, it should only be called from a worker thread. This method returns `null` when called from the main thread.

This method is not thread-safe, calling it from multiple threads at the same time will result in undefined behavior. If the calling thread is interrupted while the invocation is in-flight, it will eventually terminate and return `null` .

Note: If the provided `alias` is of an existing alias, all former grants that apps have been given to access the key and certificates associated with this alias will be revoked.

Attestation: to enable attestation, set an attestation challenge in `keySpec` via `KeyGenParameterSpec.Builder.setAttestationChallenge` . By specifying flags to the `idAttestationFlags` parameter, it is possible to request the device's unique identity to be included in the attestation record.

Specific identifiers can be included in the attestation record, and an individual attestation certificate can be used to sign the attestation record. To find out if the device supports these features, refer to `isDeviceIdAttestationSupported()`

and `isUniqueDeviceAttestationSupported()` .

Device owner, profile owner, their delegated certificate installer and the credential management app can use `ID_TYPE_BASE_INFO` to request inclusion of the general device information including manufacturer, model, brand, device and product in the attestation record. Only device owner, profile owner on an organization-owned device or affiliated user, and their delegated certificate installers can use `ID_TYPE_SERIAL` , `ID_TYPE_IMEI` and `ID_TYPE_MEID` to request unique device identifiers to be attested (the serial number, IMEI and MEID correspondingly), if supported by the device (see `isDeviceIdAttestationSupported()`). Additionally, device owner, profile owner on an organization-owned device and their delegated certificate installers can also request the attestation record to be signed using an individual attestation certificate by specifying the `ID_TYPE_INDIVIDUAL_ATTESTATION` flag (if supported by the device, see `isUniqueDeviceAttestationSupported()`).

If any of `ID_TYPE_SERIAL` , `ID_TYPE_IMEI` and `ID_TYPE_MEID` is set, it is implicitly assumed that `ID_TYPE_BASE_INFO` is also set.

Attestation using `ID_TYPE_INDIVIDUAL_ATTESTATION` can only be requested if key generation is done in StrongBox.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with, or `null` if the caller is not a device admin.
`algorithm`	`String` : The key generation algorithm, see `KeyPairGenerator` . This value cannot be `null` .
`keySpec`	`KeyGenParameterSpec` : Specification of the key to generate, see `KeyPairGenerator` . This value cannot be `null` .
`id Attestation Flags`	`int` : A bitmask of the identifiers that should be included in the attestation record (`ID_TYPE_BASE_INFO` , `ID_TYPE_SERIAL` , `ID_TYPE_IMEI` and `ID_TYPE_MEID`), and `ID_TYPE_INDIVIDUAL_ATTESTATION` if the attestation record should be signed using an individual attestation certificate. `0` should be passed in if no device identification is required in the attestation record and the batch attestation certificate should be used. If any flag is specified, then an attestation challenge must be included in the `keySpec` . Value is either `0` or a combination of the following: • `ID_TYPE_BASE_INFO` • `ID_TYPE_SERIAL` • `ID_TYPE_IMEI` • `ID_TYPE_MEID` • `ID_TYPE_INDIVIDUAL_ATTESTATION`

Returns	
`AttestedKeyPair`	A non-null `AttestedKeyPair` if the key generation succeeded, null otherwise.

Throws	
StrongBox Unavailable Exception	if the use of StrongBox for key generation was specified in `keySpec` but the device does not have one.
Illegal Argument Exception	in the following cases: • The alias in `keySpec` is empty. • The algorithm specification in `keySpec` is not `RSAKeyGenParameterSpec` or `ECGenParameterSpec`. • Device ID attestation was requested but the `keySpec` does not contain an attestation challenge.
Security Exception	if `admin` is not `null` and not a device or profile owner, or `admin` is null but the calling application is not a delegated certificate installer or credential management app. If Device ID attestation is requested (using `ID_TYPE_SERIAL`, `ID_TYPE_IMEI` or `ID_TYPE_MEID`), the caller must be the Device Owner or the Certificate Installer delegate.
Unsupported Operation Exception	if Device ID attestation or individual attestation was requested but the underlying hardware does not support it.

getAccountTypesWithManagementDisabled

```
public String[] getAccountTypesWithManagementDisabled ()
```

Gets the array of accounts for which account management is disabled by the profile owner or device owner.

Account management can be disabled/enabled by calling `setAccountManagementDisabled(ComponentName, String, boolean)`.

This method may be called on the `DevicePolicyManager` instance returned from `getParentProfileInstance(ComponentName)`. Note that only a profile owner on an organization-owned device can affect account types on the parent profile instance.

Returns	
String[]	a list of account types for which account management has been disabled. This value may be `null`.

getActiveAdmins

```
public List<ComponentName> getActiveAdmins ()
```

Return a list of all currently active device administrators' component names. If there are no administrators `null` may be returned.

Returns	
`List<ComponentName>`	

getAffiliationIds

```
public Set<String> getAffiliationIds (ComponentName admin)
```

Returns the set of affiliation ids previously set via `setAffiliationIds(ComponentName, Set)`, or an empty set if none have been set.

Parameters	
`admin`	`ComponentName`: This value cannot be `null`.

Returns	
`Set<String>`	This value cannot be `null`.

getAlwaysOnVpnLockdownWhitelist

```
public Set<String> getAlwaysOnVpnLockdownWhitelist (ComponentName admin)
```

Called by device or profile owner to query the set of packages that are allowed to access the network directly when always-on VPN is in lockdown mode but not connected. Returns `null` when always-on VPN is not active or not in lockdown mode.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null`.

Returns	
`Set<String>`	

Throws	
`SecurityException`	if `admin` is not a device or a profile owner.

getAlwaysOnVpnPackage

```
public String getAlwaysOnVpnPackage (ComponentName admin)
```

Called by a device or profile owner to read the name of the package administering an always-on VPN connection for the current user. If there is no such package, or the always-on VPN is provided by the system instead of by an application, `null` will be returned.

Parameters	
`admin`	`ComponentName` : This value cannot be `null` .
Returns	
`String`	Package name of VPN controller responsible for always-on VPN, or `null` if none is set.
Throws	
`SecurityException`	if `admin` is not a device or a profile owner.

getAppFunctionsPolicy

```
public int getAppFunctionsPolicy ()
```

Returns the current `AppFunctionManager` policy.

The returned policy will be the current resolved policy rather than the policy set by the calling admin.

Returns	
`int`	Value is one of the following: • `APP_FUNCTIONS_NOT_CONTROLLED_BY_POLICY` • `APP_FUNCTIONS_DISABLED` • `APP_FUNCTIONS_DISABLED_CROSS_PROFILE`
Throws	
`SecurityException`	if caller is not a device owner, a profile owner or a holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_APP_FUNCTIONS` .

getApplicationRestrictions

```
public Bundle getApplicationRestrictions (ComponentName admin,
                String packageName)
```

Retrieves the application restrictions for a given target application running in the calling user.

The caller must be a profile or device owner on that user, or the package allowed to manage application restrictions via `setDelegatedScopes(ComponentName, String, List)` with the `DELEGATION_APP_RESTRICTIONS` scope; otherwise a security exception will be thrown.

NOTE: The method performs disk I/O and shouldn't be called on the main thread.
This method may take several seconds to complete, so it should only be called from a worker thread.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with, or `null` if called by the application restrictions managing package.
`packageName`	`String` : The name of the package to fetch restricted settings of.
Returns	
`Bundle`	`Bundle` of settings corresponding to what was set last time `DevicePolicyManager.setApplicationRestrictions` was called, or an empty `Bundle` if no restrictions have been set. This value cannot be `null` .
Throws	
`SecurityException`	if `admin` is not a device or profile owner.

getApplicationRestrictionsManagingPackage

```
public String getApplicationRestrictionsManagingPackage (ComponentName admin)
```

This method was deprecated in API level 26.
From `Build.VERSION_CODES.O` . Use `getDelegatePackages(ComponentName, String)` with the `DELEGATION_APP_RESTRICTIONS` scope instead.

Called by a profile owner or device owner to retrieve the application restrictions managing package for the current user, or `null` if none is set. If there are multiple delegates this function will return one of them.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
Returns	
`String`	The package name allowed to manage application restrictions on the current user, or `null` if none is set.
Throws	
`SecurityException`	if `admin` is not a device or profile owner.

getAutoTimeEnabled

```
public boolean getAutoTimeEnabled (ComponentName admin)
```

Returns true if auto time is enabled on the device.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null` .
Returns	
`boolean`	true if auto time is enabled on the device.
Throws	
`SecurityException`	if caller is not a device owner, a profile owner for the primary user, or a profile owner of an organization-owned managed profile.

getAutoTimePolicy

```
public int getAutoTimePolicy ()
```

Returns current auto time policy's state.

Returns	
`int`	One of `AUTO_TIME_ENABLED` if enabled, `AUTO_TIME_DISABLED` if disabled and `AUTO_TIME_NOT_CONTROLLED_BY_POLICY` if it's not controlled by policy. Value is one of the following: • `AUTO_TIME_NOT_CONTROLLED_BY_POLICY` • `AUTO_TIME_DISABLED` • `AUTO_TIME_ENABLED`
Throws	
`SecurityException`	if caller is not a device owner, a profile owner for the primary user, or a profile owner of an organization-owned managed profile, or if the caller does not hold the required permission.

getAutoTimeRequired

```
public boolean getAutoTimeRequired ()
```

This method was deprecated in API level 30.
From `Build.VERSION_CODES.R` . Use `getAutoTimeEnabled(ComponentName)`

Returns	
`boolean`	true if auto time is required.

getAutoTimeZoneEnabled

```
public boolean getAutoTimeZoneEnabled (ComponentName admin)
```

Returns true if auto time zone is enabled on the device.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null`.

Returns	
`boolean`	true if auto time zone is enabled on the device.

Throws	
`SecurityException`	if caller is not a device owner, a profile owner for the primary user, or a profile owner of an organization-owned managed profile.

getAutoTimeZonePolicy

```
public int getAutoTimeZonePolicy ()
```

Returns auto time zone policy's current state.

Returns	
`int`	One of `AUTO_TIME_ZONE_ENABLED` if enabled, `AUTO_TIME_ZONE_DISABLED` if disabled and `AUTO_TIME_ZONE_NOT_CONTROLLED_BY_POLICY` if the state is not controlled by policy. Value is one of the following: • `AUTO_TIME_ZONE_NOT_CONTROLLED_BY_POLICY` • `AUTO_TIME_ZONE_DISABLED` • `AUTO_TIME_ZONE_ENABLED`

Throws	
`SecurityException`	if caller is not a device owner, a profile owner for the primary user, or a profile owner of an organization-owned managed profile, or if the caller does not hold the required permission.

getBindDeviceAdminTargetUsers

```
public List<UserHandle> getBindDeviceAdminTargetUsers (ComponentName admin)
```

Returns the list of target users that the calling device owner or owner of secondary user can use when calling `bindDeviceAdminServiceAsUser(ComponentName, Intent, ServiceConnection, BindServiceFlags, UserHandle)`.

A device owner can bind to a service from a secondary managed user and vice versa, provided that both users are affiliated. See `setAffiliationIds(ComponentName, Set)`.

Parameters	
`admin`	`ComponentName`: This value cannot be `null`.
Returns	
`List<UserHandle>`	This value cannot be `null`.

getBluetoothContactSharingDisabled

```
public boolean getBluetoothContactSharingDisabled (ComponentName admin)
```

Called by a profile owner of a managed profile to determine whether or not Bluetooth devices cannot access enterprise contacts.

The calling device admin must be a profile owner. If it is not, a security exception will be thrown.

This API works on managed profile only.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null`.
Returns	
`boolean`	
Throws	
`SecurityException`	if `admin` is not a profile owner.

getCameraDisabled

```
public boolean getCameraDisabled (ComponentName admin)
```

Determine whether or not the device's cameras have been disabled for this user, either by the calling admin, if specified, or all admins.

Starting with Android `Build.VERSION_CODES.CINNAMON_BUN`, this method also checks for external USB cameras that connect directly via the `UsbConstants.USB_CLASS_VIDEO` interface.

This method can be called on the `DevicePolicyManager` instance, returned by `getParentProfileInstance(ComponentName)`, where the caller must be the profile owner of an organization-owned managed profile.

Parameters	
`admin`	`ComponentName`: The name of the admin component to check, or `null` to check whether any admins have disabled the camera
Returns	
`boolean`	

getCertInstallerPackage

```
public String getCertInstallerPackage (ComponentName admin)
```

This method was deprecated in API level 26.
From `Build.VERSION_CODES.O`. Use `getDelegatePackages(ComponentName, String)` with the `DELEGATION_CERT_INSTALL` scope instead.

Called by a profile owner or device owner to retrieve the certificate installer for the user, or `null` if none is set. If there are multiple delegates this function will return one of them.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null`.
Returns	
`String`	The package name of the current delegated certificate installer, or `null` if none is set.
Throws	
`SecurityException`	if `admin` is not a device or a profile owner.

getContentProtectionPolicy

```
public int getContentProtectionPolicy (ComponentName admin)
```

Returns the current content protection policy.

The returned policy will be the current resolved policy rather than the policy set by the calling admin.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null` .
Returns	
`int`	Value is one of the following: • `CONTENT_PROTECTION_NOT_CONTROLLED_BY_POLICY` • `CONTENT_PROTECTION_DISABLED` • `CONTENT_PROTECTION_ENABLED`
Throws	
`SecurityException`	if `admin` is not the device owner, the profile owner of an affiliated user or profile, or the profile owner when no device owner is set or holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_CONTENT_PROTECTION` .

getCredentialManagerPolicy

```
public PackagePolicy getCredentialManagerPolicy ()
```

Called by a device owner or profile owner of a managed profile to retrieve the credential manager policy.

Returns	
`PackagePolicy`	the current credential manager policy if null then this policy has not been configured.
Throws	
`SecurityException`	if caller is not a device owner or profile owner of a managed profile.

getCrossProfileCalendarPackages

```
public Set<String> getCrossProfileCalendarPackages (ComponentName admin)
```

This method was deprecated in API level 34.
Use `setCrossProfilePackages(ComponentName,Set)` .

Gets a set of package names that are allowed to access cross-profile calendar APIs.

Called by a profile owner of a managed profile.

Parameters	
`admin`	`ComponentName` : which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .

Returns	
`Set<String>`	the set of names of packages that were previously allowed via `setCrossProfileCalendarPackages(ComponentName,Set)` , or an empty set if none have been allowed. This value may be `null` .

Throws	
`SecurityException`	if `admin` is not a profile owner

getCrossProfileCallerIdDisabled

```
public boolean getCrossProfileCallerIdDisabled (ComponentName admin)
```

This method was deprecated in API level 34.
starting with `Build.VERSION_CODES.UPSIDE_DOWN_CAKE` , use `getManagedProfileCallerIdAccessPolicy()` instead

Called by a profile owner of a managed profile to determine whether or not caller-Id information has been disabled.

The calling device admin must be a profile owner. If it is not, a security exception will be thrown.

Starting with `Build.VERSION_CODES.UPSIDE_DOWN_CAKE` , this will return true when `setManagedProfileCallerIdAccessPolicy(PackagePolicy)` has been set with a non-null policy whose policy type is NOT `PackagePolicy.PACKAGE_POLICY_BLOCKLIST`

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .

Returns	
`boolean`	

Throws	
`SecurityException`	if `admin` is not a profile owner.

getCrossProfileContactsSearchDisabled

```
public boolean getCrossProfileContactsSearchDisabled (ComponentName admin)
```

This method was deprecated in API level 34.
From `Build.VERSION_CODES.UPSIDE_DOWN_CAKE` use `getManagedProfileContactsAccessPolicy()`

Called by a profile owner of a managed profile to determine whether or not contacts search has been disabled.

The calling device admin must be a profile owner. If it is not, a security exception will be thrown.

Starting with `Build.VERSION_CODES.UPSIDE_DOWN_CAKE`, this will return true when `setManagedProfileContactsAccessPolicy(PackagePolicy)` has been set with a non-null policy whose policy type is NOT `PackagePolicy.PACKAGE_POLICY_BLOCKLIST`

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null`.

Returns	
`boolean`	

Throws	
`SecurityException`	if `admin` is not a profile owner.

getCrossProfilePackages

```
public Set<String> getCrossProfilePackages (ComponentName admin)
```

Returns the set of package names that the admin has previously set as allowed to request user consent for cross-profile communication, via `setCrossProfilePackages(ComponentName,Set)`.

Assumes that the caller is a profile owner and is the given `admin`.

Note that other apps not included in the returned set may be able to request user consent for cross-profile communication if they have been explicitly allowlisted by the OEM.

Parameters	
`admin`	`ComponentName`: the `DeviceAdminReceiver` this request is associated with. This value cannot be `null`.

Returns	
`Set<String>`	the set of package names the admin has previously set as allowed to request user consent for cross-profile communication, via `setCrossProfilePackages(ComponentName,Set)` This value cannot be `null`.

getCrossProfileWidgetProviders

```
public List<String> getCrossProfileWidgetProviders (ComponentName admin)
```

Called by the profile owner of a managed profile or a holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_PROFILE_INTERACTION` to query providers from which packages are available in the parent profile.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null` .

Returns	
`List<String>`	The allowlisted package list. This value cannot be `null` .

Throws	
`SecurityException`	if `admin` is not a profile owner and not a holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_PROFILE_INTERACTION` .

See also:

- `addCrossProfileWidgetProvider(android.content.ComponentName,String)`
- `removeCrossProfileWidgetProvider(android.content.ComponentName,String)`

getCurrentFailedPasswordAttempts

```
public int getCurrentFailedPasswordAttempts ()
```

Retrieve the number of times the user has failed at entering a password since that last successful password entry.

This method can be called on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to retrieve the number of failed password attempts for the parent user.

The calling device admin must have requested `DeviceAdminInfo.USES_POLICY_WATCH_LOGIN` to be able to call this method; if it has not, a security exception will be thrown.

On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, the password is always empty and this method always returns 0.
Requires the `PackageManager#FEATURE_SECURE_LOCK_SCREEN` feature which can be detected using `PackageManager.hasSystemFeature(String)` .

Returns

`int`	The number of times user has entered an incorrect password since the last correct password entry.

Throws	
`SecurityException`	if the calling application does not own an active administrator that uses `DeviceAdminInfo.USES_POLICY_WATCH_LOGIN`

getDelegatePackages

```
public List<String> getDelegatePackages (ComponentName admin,
                String delegationScope)
```

Called by a profile owner or device owner to retrieve a list of delegate packages that were granted a delegation scope.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null`.
`delegationScope`	`String`: The scope whose delegates should be retrieved. This value cannot be `null`.

Returns	
`List<String>`	A list of package names of the current delegated packages for `delegationScope`. This value may be `null`.

Throws	
`SecurityException`	if `admin` is not a device or a profile owner.

getDelegatedScopes

```
public List<String> getDelegatedScopes (ComponentName admin,
                String delegatedPackage)
```

Called by a profile owner or device owner to retrieve a list of the scopes given to a delegate package. Other apps can use this method to retrieve their own delegated scopes by passing `null` for `admin` and their own package name as `delegatedPackage`.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with, or `null` if the caller is `delegatedPackage`.
`delegatedPackage`	`String`: The package name of the app whose scopes should be retrieved. This value cannot be `null`.

Returns	
List<String>	A list containing the scopes given to `delegatedPackage`.

Throws	
SecurityException	if `admin` is not a device or a profile owner.

getDeviceOwnerLockScreenInfo

```
public CharSequence getDeviceOwnerLockScreenInfo ()
```

Returns	
CharSequence	The device owner information. If it is not set returns `null`.

getDevicePolicyManagementRoleHolderPackage

```
public String getDevicePolicyManagementRoleHolderPackage ()
```

Returns the package name of the device policy management role holder.

If the device policy management role holder is not configured for this device, returns `null`.

Returns	
String	

getEndUserSessionMessage

```
public CharSequence getEndUserSessionMessage (ComponentName admin)
```

Returns the user session end message.

Parameters	
`admin`	`ComponentName`: which `DeviceAdminReceiver` this request is associated with. This value cannot be `null`.

Returns	
CharSequence	

Throws	
SecurityException	if `admin` is not a device owner.

getEnrollmentSpecificId

```
public String getEnrollmentSpecificId ()
```

Returns an enrollment-specific identifier of this device, which is guaranteed to be the same value for the same device, enrolled into the same organization by the same managing app. This identifier is high-entropy, useful for uniquely identifying individual devices within the same organisation. It is available both in a work profile and on a fully-managed device. The identifier would be consistent even if the work profile is removed and enrolled again (to the same organization), or the device is factory reset and re-enrolled. Can only be called by the Profile Owner and Device Owner, and starting from Android `Build.VERSION_CODES.VANILLA_ICE_CREAM`, holders of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_CERTIFICATES`. If `setOrganizationId(String)` was not called, then the returned value will be an empty string.

Note about access to device identifiers: a device owner, a profile owner of an organization-owned device or the delegated certificate installer (holding the `DELEGATION_CERT_INSTALL` delegation) on such a device can still obtain hardware identifiers by calling e.g. `Build.getSerial()`, in addition to using this method. However, a profile owner on a personal (non organization-owned) device, or the delegated certificate installer on such a device, cannot obtain hardware identifiers anymore and must switch to using this method.

Returns	
`String`	A stable, enrollment-specific identifier. This value cannot be `null`.

Throws	
`SecurityException`	if the caller is not a profile owner, device owner or holding the `Manifest.permission.MANAGE_DEVICE_POLICY_CERTIFICATES` permission

getFactoryResetProtectionPolicy

```
public FactoryResetProtectionPolicy getFactoryResetProtectionPolicy (ComponentName admin)
```

Callable by device owner or profile owner of an organization-owned device, to retrieve the current factory reset protection (FRP) policy set previously by `setFactoryResetProtectionPolicy(ComponentName, FactoryResetProtectionPolicy)`.

This method can also be called by the FRP management agent on device or with the permission `Manifest.permission.MASTER_CLEAR`, in which case, it can pass `null` as the ComponentName.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with or `null` if the caller is not a device admin

Returns

FactoryResetProtectionPolicy	The current FRP policy object or `null` if no policy is set.
Throws	
SecurityException	if `admin` is not a device owner, a profile owner of an organization-owned device or the FRP management agent.
UnsupportedOperationException	if factory reset protection is not supported on the device.

getGlobalPrivateDnsHost

```
public String getGlobalPrivateDnsHost (ComponentName admin)
```

Returns the system-wide Private DNS host.

Parameters	
`admin`	`ComponentName`: which DeviceAdminReceiver this request is associated with. This value cannot be `null`.
Returns	
String	The hostname used for Private DNS queries, null if none is set.
Throws	
SecurityException	if the caller is not the device owner.

getGlobalPrivateDnsMode

```
public int getGlobalPrivateDnsMode (ComponentName admin)
```

Returns the system-wide Private DNS mode.

Parameters	
`admin`	`ComponentName`: which DeviceAdminReceiver this request is associated with. This value cannot be `null`.
Returns	
`int`	one of `PRIVATE_DNS_MODE_OFF`, `PRIVATE_DNS_MODE_OPPORTUNISTIC`, `PRIVATE_DNS_MODE_PROVIDER_HOSTNAME` or `PRIVATE_DNS_MODE_UNKNOWN`.
Throws	
SecurityException	if the caller is not the device owner.

getInstalledCaCerts

```
public List<byte[]> getInstalledCaCerts (ComponentName admin)
```

Returns all CA certificates that are currently trusted, excluding system CA certificates. If a user has installed any certificates by other means than device policy these will be included too.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with, or `null` if calling from a delegated certificate installer.

Returns	
`List<byte[]>`	a List of byte[] arrays, each encoding one user CA certificate. This value cannot be `null` .

Throws	
`SecurityException`	if `admin` is not `null` and not a device or profile owner.

getKeepUninstalledPackages

```
public List<String> getKeepUninstalledPackages (ComponentName admin)
```

Get the list of apps to keep around as APKs even if no user has currently installed it. This function can be called by a device owner or by a delegate given the `DELEGATION_KEEP_UNINSTALLED_PACKAGES` scope via `setDelegatedScopes(ComponentName, String, List)` .

Please note that packages returned in this method are not automatically pre-cached.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with, or `null` if the caller is a keep uninstalled packages delegate.

Returns	
`List<String>`	List of package names to keep cached. This value may be `null` .

getKeyPairGrants

```
public Map<Integer, Set<String>> getKeyPairGrants (String alias)
```

Called by a device or profile owner, or delegated certificate chooser (an app that has been delegated the `DELEGATION_CERT_SELECTION` privilege), to query which apps have access to a given KeyChain key. Key are granted on

a per-UID basis, so if several apps share the same UID, granting access to one of them automatically grants it to others. This method returns a map containing one entry per grantee UID. Entries have UIDs as keys and sets of corresponding package names as values. In particular, grantee packages that don't share UID with other packages are represented by entries having singleton sets as values.

Parameters	
`alias`	`String` : The alias of the key to grant access to. This value cannot be `null` .
Returns	
`Map<Integer, Set<String>>`	apps that have access to a given key, arranged in a map from UID to sets of package names. This value cannot be `null` .
Throws	
`IllegalArgumentException`	if `alias` doesn't correspond to an existing key.
`SecurityException`	if the caller is not a device owner, a profile owner or delegated certificate chooser.

getKeyguardDisabledFeatures

```
public int getKeyguardDisabledFeatures (ComponentName admin)
```

Determine whether or not features have been disabled in keyguard either by the calling admin, if specified, or all admins that set restrictions on this user and its participating profiles. Restrictions on profiles that have a separate challenge are not taken into account.

This method can be called on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to retrieve restrictions on the parent profile.

Parameters	
`admin`	`ComponentName` : The name of the admin component to check, or `null` to check whether any admins have disabled features in keyguard.
Returns	
`int`	bitfield of flags. See `setKeyguardDisabledFeatures(ComponentName,int)` for a list.

getLockTaskFeatures

```
public int getLockTaskFeatures (ComponentName admin)
```

Gets which system features are enabled for LockTask mode.

Starting from `Build.VERSION_CODES.UPSIDE_DOWN_CAKE`, the returned policy will be the current resolved policy rather than the policy set by the calling admin.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null`.
Returns	
`int`	bitfield of flags. See `setLockTaskFeatures(ComponentName,int)` for a list. Value is either `0` or a combination of the following: • `LOCK_TASK_FEATURE_NONE` • `LOCK_TASK_FEATURE_SYSTEM_INFO` • `LOCK_TASK_FEATURE_NOTIFICATIONS` • `LOCK_TASK_FEATURE_HOME` • `LOCK_TASK_FEATURE_OVERVIEW` • `LOCK_TASK_FEATURE_GLOBAL_ACTIONS` • `LOCK_TASK_FEATURE_KEYGUARD` • `LOCK_TASK_FEATURE_BLOCK_ACTIVITY_START_IN_TASK`
Throws	
`SecurityException`	if `admin` is not the device owner, the profile owner of an affiliated user or profile, or the profile owner when no device owner is set or holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_LOCK_TASK`.

getLockTaskPackages

```
public String[] getLockTaskPackages (ComponentName admin)
```

Returns the list of packages allowed to start the lock task mode.

Starting from `Build.VERSION_CODES.UPSIDE_DOWN_CAKE`, the returned policy will be the current resolved policy rather than the policy set by the calling admin.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null`.
Returns	
`String[]`	This value cannot be `null`.

Throws	
SecurityException	if admin is not the device owner, the profile owner of an affiliated user or profile, or the profile owner when no device owner is set or holder of the permission Manifest.permission.MANAGE_DEVICE_POLICY_LOCK_TASK.

getLongSupportMessage

```
public CharSequence getLongSupportMessage (ComponentName admin)
```

Called by a device admin to get the long support message.

Parameters	
admin	ComponentName: Which DeviceAdminReceiver this request is associated with. This value cannot be null.
Returns	
CharSequence	The message set by setLongSupportMessage(ComponentName,CharSequence) or null if no message has been set.
Throws	
SecurityException	if admin is not an active administrator.

getManagedProfileCallerIdAccessPolicy

```
public PackagePolicy getManagedProfileCallerIdAccessPolicy ()
```

Called by a profile owner of a managed profile to retrieve the caller id policy.

The calling device admin must be a profile owner of a managed profile. If it is not, a SecurityException will be thrown.

Returns	
PackagePolicy	the current caller id policy. This value may be null.
Throws	
SecurityException	if caller is not a profile owner of a managed profile.

getManagedProfileContactsAccessPolicy

```
public PackagePolicy getManagedProfileContactsAccessPolicy ()
```

Called by a profile owner of a managed profile to determine the current policy applied to managed profile contacts.

The calling device admin must be a profile owner of a managed profile. If it is not, a `SecurityException` will be thrown.

Returns	
`PackagePolicy`	the current contacts search policy. This value may be `null`.
Throws	
`SecurityException`	if caller is not a profile owner of a managed profile.

getManagedProfileMaximumTimeOff

```
public long getManagedProfileMaximumTimeOff (ComponentName admin)
```

Called by a profile owner of an organization-owned managed profile to get maximum time the profile is allowed to be turned off.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null`.
Returns	
`long`	Maximum time the profile is allowed to be off in milliseconds or 0 if not limited.

getManagedSubscriptionsPolicy

```
public ManagedSubscriptionsPolicy getManagedSubscriptionsPolicy ()
```

Returns the current `ManagedSubscriptionsPolicy`. If the policy has not been set, it will return a default policy of Type `ManagedSubscriptionsPolicy.TYPE_ALL_PERSONAL_SUBSCRIPTIONS`.

Returns	
`ManagedSubscriptionsPolicy`	This value cannot be `null`.

getMaximumFailedPasswordsForWipe

```
public int getMaximumFailedPasswordsForWipe (ComponentName admin)
```

Retrieve the current maximum number of login attempts that are allowed before the device or profile is wiped, for a particular admin or all admins that set restrictions on this user and its participating profiles. Restrictions on profiles that have a separate challenge are not taken into account.

This method can be called on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to retrieve the value for the parent profile.

On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, the password is always empty and this method returns a default value (0) indicating that the policy is not set.
Requires the `PackageManager#FEATURE_SECURE_LOCK_SCREEN` feature which can be detected using `PackageManager.hasSystemFeature(String)` .

Parameters	
`admin`	`ComponentName` : The name of the admin component to check, or `null` to aggregate all admins.
Returns	
`int`	

getMaximumTimeToLock

```
public long getMaximumTimeToLock (ComponentName admin)
```

Retrieve the current maximum time to unlock for a particular admin or all admins that set restrictions on this user and its participating profiles. Restrictions on profiles that have a separate challenge are not taken into account.

This method can be called on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to retrieve restrictions on the parent profile.

Parameters	
`admin`	`ComponentName` : The name of the admin component to check, or `null` to aggregate all admins.
Returns	
`long`	time in milliseconds for the given admin or the minimum value (strictest) of all admins if admin is null. Returns 0 if there are no restrictions.

getMeteredDataDisabledPackages

```
public List<String> getMeteredDataDisabledPackages (ComponentName admin)
```

Called by a device or profile owner to retrieve the list of packages which are restricted by the admin from using metered data.

Parameters	
`admin`	`ComponentName` : which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
Returns	
`List<String>`	the list of restricted package names. This value cannot be `null` .
Throws	
`SecurityException`	if `admin` is not a device or profile owner.

getMinimumRequiredWifiSecurityLevel

```
public int getMinimumRequiredWifiSecurityLevel ()
```

Returns the current Wi-Fi minimum security level.

Returns	
`int`	Value is one of the following: • `WIFI_SECURITY_OPEN` • `WIFI_SECURITY_PERSONAL` • `WIFI_SECURITY_ENTERPRISE_EAP` • `WIFI_SECURITY_ENTERPRISE_192`

getMtePolicy

```
public int getMtePolicy ()
```

Called by a device owner, profile owner of an organization-owned device to get the Memory Tagging Extension (MTE) policy Learn more about MTE

Returns

`int`	the currently set MTE policy. Value is one of the following: • `MTE_ENABLED` • `MTE_DISABLED` • `MTE_NOT_CONTROLLED_BY_POLICY`

Throws	
`SecurityException`	if caller is not permitted to set Mte policy

getNearbyAppStreamingPolicy

```
public int getNearbyAppStreamingPolicy ()
```

Returns the current runtime nearby app streaming policy set by the device or profile owner.

The caller must be the target user's device owner/profile owner or hold the `READ_NEARBY_STREAMING_POLICY` permission.

Returns	
`int`	Value is one of the following: • `NEARBY_STREAMING_NOT_CONTROLLED_BY_POLICY` • `NEARBY_STREAMING_DISABLED` • `NEARBY_STREAMING_ENABLED` • `NEARBY_STREAMING_SAME_MANAGED_ACCOUNT_ONLY`

getNearbyNotificationStreamingPolicy

```
public int getNearbyNotificationStreamingPolicy ()
```

Returns the current runtime nearby notification streaming policy set by the device or profile owner.

The caller must be the target user's device owner/profile owner or hold the `READ_NEARBY_STREAMING_POLICY` permission.

Returns	
`int`	Value is one of the following: • `NEARBY_STREAMING_NOT_CONTROLLED_BY_POLICY` • `NEARBY_STREAMING_DISABLED` • `NEARBY_STREAMING_ENABLED`

	• `NEARBY_STREAMING_SAME_MANAGED_ACCOUNT_ONLY`

getOrganizationColor

```
public int getOrganizationColor (ComponentName admin)
```

This method was deprecated in API level 31.
From `Build.VERSION_CODES.R`, the organization color is never used as the background color of the confirm credentials screen.

Called by a profile owner of a managed profile to retrieve the color used for customization. This color is used as background color of the confirm credentials screen for that user.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null`.
Returns	
`int`	The 24bit (0xRRGGBB) representation of the color to be used.
Throws	
`SecurityException`	if `admin` is not a profile owner.

getOrganizationName

```
public CharSequence getOrganizationName (ComponentName admin)
```

Called by the device owner (since API 26) or profile owner (since API 24) or holders of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_ORGANIZATION_IDENTITY` to retrieve the name of the organization under management.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null`.
Returns	
`CharSequence`	The organization name or `null` if none is set.
Throws	

SecurityException	if admin if admin is not a device or profile owner or holder of the permission Manifest.permission.MANAGE_DEVICE_POLICY_ORGANIZATION_IDENTITY.

getOverrideApns

```
public List<ApnSetting> getOverrideApns (ComponentName admin)
```

Called by device owner or managed profile owner to get all override APNs inserted by device owner or managed profile owner previously using addOverrideApn(ComponentName, ApnSetting).

Parameters	
admin	ComponentName: which DeviceAdminReceiver this request is associated with. This value cannot be null.

Returns	
List<ApnSetting>	A list of override APNs inserted by device owner.

Throws	
SecurityException	if admin is not a device owner.

getParentProfileInstance

```
public DevicePolicyManager getParentProfileInstance (ComponentName admin)
```

Called by the profile owner of a managed profile or other apps in a managed profile to obtain a DevicePolicyManager whose calls act on the parent profile.

The following methods are supported for the parent instance, all other methods will throw a SecurityException when called on the parent instance:

- getPasswordQuality(ComponentName)
- setPasswordQuality(ComponentName, int)
- getPasswordMinimumLength(ComponentName)
- setPasswordMinimumLength(ComponentName, int)
- getPasswordMinimumUpperCase(ComponentName)
- setPasswordMinimumUpperCase(ComponentName, int)
- getPasswordMinimumLowerCase(ComponentName)
- setPasswordMinimumLowerCase(ComponentName, int)
- getPasswordMinimumLetters(ComponentName)
- setPasswordMinimumLetters(ComponentName, int)
- getPasswordMinimumNumeric(ComponentName)
- setPasswordMinimumNumeric(ComponentName, int)
- getPasswordMinimumSymbols(ComponentName)

- `setPasswordMinimumSymbols(ComponentName, int)`
- `getPasswordMinimumNonLetter(ComponentName)`
- `setPasswordMinimumNonLetter(ComponentName, int)`
- `getPasswordHistoryLength(ComponentName)`
- `setPasswordHistoryLength(ComponentName, int)`
- `getPasswordExpirationTimeout(ComponentName)`
- `setPasswordExpirationTimeout(ComponentName, long)`
- `getPasswordExpiration(ComponentName)`
- `getPasswordMaximumLength(int)`
- `isActivePasswordSufficient()`
- `getCurrentFailedPasswordAttempts()`
- `getMaximumFailedPasswordsForWipe(ComponentName)`
- `setMaximumFailedPasswordsForWipe(ComponentName, int)`
- `getMaximumTimeToLock(ComponentName)`
- `setMaximumTimeToLock(ComponentName, long)`
- `lockNow()`
- `getKeyguardDisabledFeatures(ComponentName)`
- `setKeyguardDisabledFeatures(ComponentName, int)`
- `getTrustAgentConfiguration(ComponentName, ComponentName)`
- `setTrustAgentConfiguration(ComponentName, ComponentName, PersistableBundle)`
- `getRequiredStrongAuthTimeout(ComponentName)`
- `setRequiredStrongAuthTimeout(ComponentName, long)`
- `getAccountTypesWithManagementDisabled()`
- `setRequiredPasswordComplexity(int)`
- `getRequiredPasswordComplexity()`

The following methods are supported for the parent instance but can only be called by the profile owner on an organization owned managed profile:

- `getPasswordComplexity()`
- `setCameraDisabled(ComponentName, boolean)`
- `getCameraDisabled(ComponentName)`
- `setAccountManagementDisabled(ComponentName,String,boolean)`
- `setPermittedInputMethods(ComponentName, List)`
- `getPermittedInputMethods(ComponentName)`
- `wipeData(int)`

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with or `null` if the caller is not a profile owner.

Returns	
`DevicePolicyManager`	a new instance of `DevicePolicyManager` that acts on the parent profile. This value cannot be `null` .

Throws	
`SecurityException`	if the current user is not a managed profile.

getPasswordComplexity

```
public int getPasswordComplexity ()
```

Returns how complex the current user's screen lock is.

Note that when called from a profile which uses an unified challenge with its parent, the screen lock complexity of the parent will be returned.

Apps need the `permission.REQUEST_PASSWORD_COMPLEXITY` permission to call this method. On Android `Build.VERSION_CODES.S` and above, the calling application does not need this permission if it is a device owner or a profile owner.

This method can be called on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to retrieve restrictions on the parent profile.

Returns	
`int`	Value is one of the following: • `PASSWORD_COMPLEXITY_NONE` • `PASSWORD_COMPLEXITY_LOW` • `PASSWORD_COMPLEXITY_MEDIUM` • `PASSWORD_COMPLEXITY_HIGH`

Throws	
`IllegalStateException`	if the user is not unlocked.
`SecurityException`	if the calling application does not have the permission `permission.REQUEST_PASSWORD_COMPLEXITY`, and is not a device owner or a profile owner.

getPasswordExpiration

```
public long getPasswordExpiration (ComponentName admin)
```

Get the current password expiration time for a particular admin or all admins that set restrictions on this user and its participating profiles. Restrictions on profiles that have a separate challenge are not taken into account. If admin is `null`, then a composite of all expiration times is returned - which will be the minimum of all of them.

This method can be called on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to retrieve the password expiration for the parent profile.

On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, the password expiration is always disabled and this method always returns 0.
Requires the `PackageManager#FEATURE_SECURE_LOCK_SCREEN` feature which can be detected using `PackageManager.hasSystemFeature(String)`.

Parameters	
`admin`	`ComponentName`: The name of the admin component to check, or `null` to aggregate all admins.
Returns	
`long`	The password expiration time, in milliseconds since epoch.

getPasswordExpirationTimeout

```
public long getPasswordExpirationTimeout (ComponentName admin)
```

Get the password expiration timeout for the given admin. The expiration timeout is the recurring expiration timeout provided in the call to `setPasswordExpirationTimeout(ComponentName,long)` for the given admin or the aggregate of all participating policy administrators if `admin` is null. Admins that have set restrictions on profiles that have a separate challenge are not taken into account.

This method can be called on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to retrieve restrictions on the parent profile.

On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, the password expiration is always disabled and this method always returns 0.
Requires the `PackageManager#FEATURE_SECURE_LOCK_SCREEN` feature which can be detected using `PackageManager.hasSystemFeature(String)`.

Parameters	
`admin`	`ComponentName`: The name of the admin component to check, or `null` to aggregate all admins.
Returns	
`long`	The timeout for the given admin or the minimum of all timeouts

getPasswordHistoryLength

```
public int getPasswordHistoryLength (ComponentName admin)
```

Retrieve the current password history length for a particular admin or all admins that set restrictions on this user and its participating profiles. Restrictions on profiles that have a separate challenge are not taken into account.

This method can be called on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to retrieve restrictions on the parent profile.

On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, the password history length is always 0.
Requires the `PackageManager#FEATURE_SECURE_LOCK_SCREEN` feature which can be detected using `PackageManager.hasSystemFeature(String)`.

Parameters	
`admin`	`ComponentName`: The name of the admin component to check, or `null` to aggregate all admins.

Returns	
`int`	The length of the password history

getPasswordMaximumLength

```
public int getPasswordMaximumLength (int quality)
```

Return the maximum password length that the device supports for a particular password quality.

On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, the password is always empty and this method always returns 0.

Parameters	
`quality`	`int`: The quality being interrogated.

Returns	
`int`	Returns the maximum length that the user can enter.

getPasswordMinimumLength

```
public int getPasswordMinimumLength (ComponentName admin)
```

This method was deprecated in API level 31.
see `setPasswordQuality(ComponentName,int)` for details.

Retrieve the current minimum password length for a particular admin or all admins that set restrictions on this user and its participating profiles. Restrictions on profiles that have a separate challenge are not taken into account.

On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, the password is always treated as empty.

This method can be called on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to retrieve restrictions on the parent profile.

Parameters	
`admin`	`ComponentName` : The name of the admin component to check, or `null` to aggregate all admins.

Returns	
`int`	

getPasswordMinimumLetters

```
public int getPasswordMinimumLetters (ComponentName admin)
```

This method was deprecated in API level 31.
see `setPasswordQuality(ComponentName,int)` for details.

Retrieve the current number of letters required in the password for a particular admin or all admins that set restrictions on this user and its participating profiles. Restrictions on profiles that have a separate challenge are not taken into account. This is the same value as set by `setPasswordMinimumLetters(ComponentName,int)` and only applies when the password quality is `PASSWORD_QUALITY_COMPLEX` .

On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, the password is always treated as empty.

This method can be called on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to retrieve restrictions on the parent profile.

Parameters	
`admin`	`ComponentName` : The name of the admin component to check, or `null` to aggregate all admins.

Returns	
`int`	The minimum number of letters required in the password.

getPasswordMinimumLowerCase

```
public int getPasswordMinimumLowerCase (ComponentName admin)
```

This method was deprecated in API level 31.
see `setPasswordQuality(ComponentName,int)` for details.

Retrieve the current number of lower case letters required in the password for a particular admin or all admins that set restrictions on this user and its participating profiles. Restrictions on profiles that have a separate challenge are not taken into account. This is the same value as set by `setPasswordMinimumLowerCase(ComponentName,int)` and only applies when the password quality is `PASSWORD_QUALITY_COMPLEX` .

On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, the password is always treated as empty.

This method can be called on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to retrieve restrictions on the parent profile.

Parameters	
`admin`	`ComponentName` : The name of the admin component to check, or `null` to aggregate all admins.
Returns	
`int`	The minimum number of lower case letters required in the password.

getPasswordMinimumNonLetter

```
public int getPasswordMinimumNonLetter (ComponentName admin)
```

This method was deprecated in API level 31.
see `setPasswordQuality(ComponentName,int)` for details.

Retrieve the current number of non-letter characters required in the password for a particular admin or all admins that set restrictions on this user and its participating profiles. Restrictions on profiles that have a separate challenge are not taken into account. This is the same value as set by `setPasswordMinimumNonLetter(ComponentName,int)` and only applies when the password quality is `PASSWORD_QUALITY_COMPLEX` .

On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, the password is always treated as empty.

This method can be called on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to retrieve restrictions on the parent profile.

Parameters	
`admin`	`ComponentName` : The name of the admin component to check, or `null` to aggregate all admins.
Returns	
`int`	The minimum number of letters required in the password.

getPasswordMinimumNumeric

```
public int getPasswordMinimumNumeric (ComponentName admin)
```

This method was deprecated in API level 31.
see `setPasswordQuality(ComponentName,int)` for details.

Retrieve the current number of numerical digits required in the password for a particular admin or all admins that set restrictions on this user and its participating profiles. Restrictions on profiles that have a separate challenge are not taken into account. This is the same value as set by `setPasswordMinimumNumeric(ComponentName,int)` and only applies when the password quality is `PASSWORD_QUALITY_COMPLEX`.

On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, the password is always treated as empty.

This method can be called on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to retrieve restrictions on the parent profile.

Parameters	
`admin`	`ComponentName` : The name of the admin component to check, or `null` to aggregate all admins.
Returns	
`int`	The minimum number of numerical digits required in the password.

getPasswordMinimumSymbols

```
public int getPasswordMinimumSymbols (ComponentName admin)
```

This method was deprecated in API level 31.
see `setPasswordQuality(ComponentName,int)` for details.

Retrieve the current number of symbols required in the password for a particular admin or all admins that set restrictions on this user and its participating profiles. Restrictions on profiles that have a separate challenge are not taken into account. This is the same value as set by `setPasswordMinimumSymbols(ComponentName,int)` and only applies when the password quality is `PASSWORD_QUALITY_COMPLEX`.

On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, the password is always treated as empty.

This method can be called on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to retrieve restrictions on the parent profile.

Parameters	
`admin`	`ComponentName` : The name of the admin component to check, or `null` to aggregate all admins.
Returns	
`int`	The minimum number of symbols required in the password.

getPasswordMinimumUpperCase

```
public int getPasswordMinimumUpperCase (ComponentName admin)
```

This method was deprecated in API level 31.
see `setPasswordQuality(ComponentName,int)` for details.

Retrieve the current number of upper case letters required in the password for a particular admin or all admins that set restrictions on this user and its participating profiles. Restrictions on profiles that have a separate challenge are not taken into account. This is the same value as set by `setPasswordMinimumUpperCase(ComponentName,int)` and only applies when the password quality is `PASSWORD_QUALITY_COMPLEX`.

On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, the password is always treated as empty.

This method can be called on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to retrieve restrictions on the parent profile.

Parameters	
`admin`	`ComponentName`: The name of the admin component to check, or `null` to aggregate all admins.
Returns	
`int`	The minimum number of upper case letters required in the password.

getPasswordQuality

```
public int getPasswordQuality (ComponentName admin)
```

This method was deprecated in API level 31.
see `setPasswordQuality(ComponentName,int)` for details.

Retrieve the current minimum password quality for a particular admin or all admins that set restrictions on this user and its participating profiles. Restrictions on profiles that have a separate challenge are not taken into account.

This method can be called on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to retrieve restrictions on the parent profile.

Note: on devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, the password is always treated as empty.

Parameters	
`admin`	`ComponentName`: The name of the admin component to check, or `null` to aggregate all admins.
Returns	
`int`	

getPendingSystemUpdate

```
public SystemUpdateInfo getPendingSystemUpdate (ComponentName admin)
```

Get information about a pending system update. Can be called by device or profile owners, and starting from Android `Build.VERSION_CODES.VANILLA_ICE_CREAM`, holders of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_QUERY_SYSTEM_UPDATES`.

Parameters	
`admin`	`ComponentName`: Which profile or device owner this request is associated with. This value may be `null`.
Returns	
`SystemUpdateInfo`	Information about a pending system update or `null` if no update pending.
Throws	
`SecurityException`	if `admin` is not a device, profile owner or holders of `Manifest.permission.MANAGE_DEVICE_POLICY_QUERY_SYSTEM_UPDATES`.

getPermissionGrantState

```
public int getPermissionGrantState (ComponentName admin,
                String packageName,
                String permission)
```

Returns the current grant state of a runtime permission for a specific application. This function can be called by a device owner, profile owner, or by a delegate given the `DELEGATION_PERMISSION_GRANT` scope via `setDelegatedScopes(ComponentName, String, List)`.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null`.
`packageName`	`String`: The application to check the grant state for. This value cannot be `null`.
`permission`	`String`: The permission to check for. This value cannot be `null`.
Returns	
`int`	the current grant state specified by device policy. If admins have not set a grant has not set a grant state, the return value is `PERMISSION_GRANT_STATE_DEFAULT`. This does not indicate whether or not the

	permission is currently granted for the package. If a grant state was set by the profile or device owner, then the return value will be one of `PERMISSION_GRANT_STATE_DENIED` or `PERMISSION_GRANT_STATE_GRANTED`, which indicates if the permission is currently denied or granted. Value is one of the following: • `PERMISSION_GRANT_STATE_DEFAULT` • `PERMISSION_GRANT_STATE_GRANTED` • `PERMISSION_GRANT_STATE_DENIED`

Throws	
`SecurityException`	if `admin` is not a device or profile owner.

See also:

- `setPermissionGrantState(ComponentName,String,String,int)`
- `PackageManager.checkPermission(String,String)`
- `setDelegatedScopes(ComponentName, String, List)`
- `DELEGATION_PERMISSION_GRANT`

getPermissionPolicy

```
public int getPermissionPolicy (ComponentName admin)
```

Returns the current runtime permission policy set by the device or profile owner. The default is `PERMISSION_POLICY_PROMPT`.

Parameters	
`admin`	`ComponentName`: Which profile or device owner this request is associated with.
Returns	
`int`	the current policy for future permission requests.

getPermittedAccessibilityServices

```
public List<String> getPermittedAccessibilityServices (ComponentName admin)
```

Returns the list of permitted accessibility services set by this device or profile owner.

An empty list means no accessibility services except system services are allowed. `null` means all accessibility services are allowed.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .

Returns	
`List<String>`	List of accessibility service package names.

Throws	
`SecurityException`	if `admin` is not a device or profile owner.

getPermittedCrossProfileNotificationListeners

```
public List<String> getPermittedCrossProfileNotificationListeners (ComponentName admin)
```

Returns the list of packages installed on the primary user that allowed to use a `NotificationListenerService` to receive notifications from this managed profile, as set by the profile owner.

An empty list means no notification listener services except system ones are allowed. A `null` return value indicates that all notification listeners are allowed.

Parameters	
`admin`	`ComponentName` : This value cannot be `null` .

Returns	
`List<String>`	

getPermittedInputMethods

```
public List<String> getPermittedInputMethods (ComponentName admin)
```

Returns the list of permitted input methods set by this device or profile owner.

This method can be called on the `DevicePolicyManager` instance, returned by `getParentProfileInstance(ComponentName)` , where the caller must be a profile owner of an organization-owned managed profile. If called on the parent instance, then the returned list of permitted input methods are those which are applied on the personal profile.

An empty list means no input methods except system input methods are allowed. Null means all input methods are allowed.

Parameters

`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null` .
Returns	
`List<String>`	List of input method package names. This value may be `null` .
Throws	
`SecurityException`	if `admin` is not a device, profile owner or if called on the parent profile and the `admin` is not a profile owner of an organization-owned managed profile.

getPersonalAppsSuspendedReasons

```
public int getPersonalAppsSuspendedReasons (ComponentName admin)
```

Called by profile owner of an organization-owned managed profile to check whether personal apps are suspended.

Parameters	
`admin`	`ComponentName` : This value cannot be `null` .
Returns	
`int`	a bitmask of reasons for personal apps suspension or `PERSONAL_APPS_NOT_SUSPENDED` if apps are not suspended. Value is either `0` or a combination of the following: • `PERSONAL_APPS_NOT_SUSPENDED` • `PERSONAL_APPS_SUSPENDED_EXPLICITLY` • `PERSONAL_APPS_SUSPENDED_PROFILE_TIMEOUT`

getPreferentialNetworkServiceConfigs

```
public List<PreferentialNetworkServiceConfig> getPreferentialNetworkServiceConfigs ()
```

Get preferential network configuration

Returns	
`List<PreferentialNetworkServiceConfig>`	preferential network configuration. This value cannot be `null` .
Throws	

`SecurityException`	if the caller is not the profile owner or device owner.

getRequiredPasswordComplexity

```
public int getRequiredPasswordComplexity ()
```

Gets the password complexity requirement set by `setRequiredPasswordComplexity(int)`, for the current user.

The difference between this method and `getPasswordComplexity()` is that this method simply returns the value set by `setRequiredPasswordComplexity(int)` while `getPasswordComplexity()` returns the complexity of the actual password.

This method can be called on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to get restrictions on the parent profile.

Returns	
`int`	Value is one of the following: • `PASSWORD_COMPLEXITY_NONE` • `PASSWORD_COMPLEXITY_LOW` • `PASSWORD_COMPLEXITY_MEDIUM` • `PASSWORD_COMPLEXITY_HIGH`

Throws	
`SecurityException`	if the calling application is not a device owner or a profile owner.

getRequiredStrongAuthTimeout

```
public long getRequiredStrongAuthTimeout (ComponentName admin)
```

Determine for how long the user will be able to use secondary, non strong auth for authentication, since last strong method authentication (password, pin or pattern) was used. After the returned timeout the user is required to use strong authentication method.

This method can be called on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to retrieve restrictions on the parent profile.

On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, 0 is returned to indicate that no timeout is configured.
Requires the `PackageManager#FEATURE_SECURE_LOCK_SCREEN` feature which can be detected using `PackageManager.hasSystemFeature(String)`.

Parameters

`admin`	`ComponentName` : The name of the admin component to check, or `null` to aggregate across all participating admins.

Returns	
`long`	The timeout in milliseconds or 0 if not configured for the provided admin.

getResources

```
public DevicePolicyResourcesManager getResources ()
```

Returns a `DevicePolicyResourcesManager` containing the required APIs to set, reset, and get device policy related resources.

Returns	
`DevicePolicyResourcesManager`	This value cannot be `null` .

getScreenCaptureDisabled

```
public boolean getScreenCaptureDisabled (ComponentName admin)
```

Determine whether or not screen capture has been disabled by the calling admin, if specified, or all admins.

This method can be called on the `DevicePolicyManager` instance, returned by `getParentProfileInstance(ComponentName)` , where the caller must be the profile owner of an organization-owned managed profile (the calling admin must be specified).

Parameters	
`admin`	`ComponentName` : The name of the admin component to check, or `null` to check whether any admins have disabled screen capture.

Returns	
`boolean`	

getSecondaryUsers

```
public List<UserHandle> getSecondaryUsers (ComponentName admin)
```

Called by a device owner to list all secondary users on the device. Managed profiles are not considered as secondary users.

Used for various user management APIs, including `switchUser(ComponentName, UserHandle)` , `removeUser(ComponentName, UserHandle)` and `stopUser(ComponentName, UserHandle)` .

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .

Returns	
`List<UserHandle>`	list of other `UserHandle` s on the device.

Throws	
`SecurityException`	if `admin` is not a device owner.

See also:

- `removeUser(ComponentName,UserHandle)`
- `switchUser(ComponentName,UserHandle)`
- `startUserInBackground(ComponentName,UserHandle)`
- `stopUser(ComponentName,UserHandle)`

getShortSupportMessage

```
public CharSequence getShortSupportMessage (ComponentName admin)
```

Called by a device admin or holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_SUPPORT_MESSAGE` to get the short support message.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null` .

Returns	
`CharSequence`	The message set by `setShortSupportMessage(ComponentName,CharSequence)` or null if no message has been set.

Throws	
`SecurityException`	if `admin` is not an active administrator and not a holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_SUPPORT_MESSAGE` ..

getStartUserSessionMessage

```
public CharSequence getStartUserSessionMessage (ComponentName admin)
```

Returns the user session start message.

Parameters	
`admin`	`ComponentName` : which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .

Returns	
`CharSequence`	

Throws	
`SecurityException`	if `admin` is not a device owner.

getStorageEncryption

```
public boolean getStorageEncryption (ComponentName admin)
```

This method was deprecated in API level 30.
This method only returns the value set by `setStorageEncryption(ComponentName, boolean)` . It does not actually reflect the storage encryption status. Use `getStorageEncryptionStatus()` for that. Called by an application that is administering the device to determine the requested setting for secure storage.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. If null, this will return the requested encryption setting as an aggregate of all active administrators.

Returns	
`boolean`	true if the admin(s) are requesting encryption, false if not.

getStorageEncryptionStatus

```
public int getStorageEncryptionStatus ()
```

Called by an application that is administering the device to determine the current encryption status of the device.

Depending on the returned status code, the caller may proceed in different ways. If the result is `ENCRYPTION_STATUS_UNSUPPORTED` , the storage system does not support encryption. If the result is `ENCRYPTION_STATUS_INACTIVE` , use `ACTION_START_ENCRYPTION` to begin the process of encrypting or decrypting the storage. If the result is `ENCRYPTION_STATUS_ACTIVE_DEFAULT_KEY` , the storage system has enabled encryption but no password is set so further action may be required. If the result is `ENCRYPTION_STATUS_ACTIVATING` , `ENCRYPTION_STATUS_ACTIVE` or `ENCRYPTION_STATUS_ACTIVE_PER_USER` , no further action is required.

Returns

`int`	current status of encryption. The value will be one of `ENCRYPTION_STATUS_UNSUPPORTED`, `ENCRYPTION_STATUS_INACTIVE`, `ENCRYPTION_STATUS_ACTIVATING`, `ENCRYPTION_STATUS_ACTIVE_DEFAULT_KEY`, `ENCRYPTION_STATUS_ACTIVE`, or `ENCRYPTION_STATUS_ACTIVE_PER_USER`.

Throws	
`SecurityException`	if called on a parent instance.

getSubscriptionIds

```
public Set<Integer> getSubscriptionIds ()
```

Returns the subscription ids of all subscriptions which were downloaded by the calling admin.

This returns only the subscriptions which were downloaded by the calling admin via `EuiccManager.downloadSubscription(DownloadableSubscription, boolean, PendingIntent)`. If a subscription is returned by this method then in it subject to management controls and cannot be removed by users.

Callable by device owners and profile owners.
Requires `Manifest.permission.MANAGE_DEVICE_POLICY_MANAGED_SUBSCRIPTIONS`

Returns	
`Set<Integer>`	ids of all managed subscriptions currently downloaded by an admin on the device. This value cannot be `null`.

Throws	
`SecurityException`	if the caller is not authorized to call this method.

getSystemUpdatePolicy

```
public SystemUpdatePolicy getSystemUpdatePolicy ()
```

Retrieve a local system update policy set previously by `setSystemUpdatePolicy(ComponentName, SystemUpdatePolicy)`.

Returns	
`SystemUpdatePolicy`	The current policy object, or `null` if no policy is set.

getTransferOwnershipBundle

```
public PersistableBundle getTransferOwnershipBundle ()
```

Returns the data passed from the current administrator to the new administrator during an ownership transfer. This is the same `bundle` passed in `transferOwnership(ComponentName,ComponentName,PersistableBundle)`. The bundle is persisted until the profile owner or device owner is removed.

This is the same `bundle` received in the `DeviceAdminReceiver.onTransferOwnershipComplete(Context,PersistableBundle)`. Use this method to retrieve it after the transfer as long as the new administrator is the active device or profile owner.

Returns `null` if no ownership transfer was started for the calling user.

Returns	
`PersistableBundle`	
Throws	
`SecurityException`	if the caller is not a device or profile owner.

See also:

- `transferOwnership(ComponentName, ComponentName, PersistableBundle)`
- `DeviceAdminReceiver.onTransferOwnershipComplete(Context,PersistableBundle)`

getTrustAgentConfiguration

```
public List<PersistableBundle> getTrustAgentConfiguration (ComponentName admin,
                ComponentName agent)
```

Gets configuration for the given trust agent based on aggregating all calls to `setTrustAgentConfiguration(ComponentName,ComponentName,PersistableBundle)` for all device admins.

This method can be called on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to retrieve the configuration set on the parent profile.

On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, null is always returned. Requires the `PackageManager#FEATURE_SECURE_LOCK_SCREEN` feature which can be detected using `PackageManager.hasSystemFeature(String)`.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. If null, this function returns a list of configurations for all admins that declare `KEYGUARD_DISABLE_TRUST_AGENTS`. If any admin declares `KEYGUARD_DISABLE_TRUST_AGENTS` but doesn't call `setTrustAgentConfiguration(ComponentName,ComponentName,PersistableBundle)` for this `agent` or calls it with a null configuration, null is returned.
`agent`	`ComponentName`: Which component to get enabled features for. This value cannot be `null`.

Returns	
List<PersistableBundle>	configuration for the given trust agent.

getUserControlDisabledPackages

```
public List<String> getUserControlDisabledPackages (ComponentName admin)
```

Returns the list of packages over which user control is disabled by a device or profile owner or holders of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_APPS_CONTROL`.

Starting from `Build.VERSION_CODES.UPSIDE_DOWN_CAKE`, the returned policy will be the current resolved policy rather than the policy set by the calling admin.

Parameters	
admin	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null`.

Returns	
List<String>	This value cannot be `null`.

Throws	
SecurityException	if `admin` is not a device or profile owner or holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_APPS_CONTROL`.

getUserRestrictions

```
public Bundle getUserRestrictions (ComponentName admin)
```

Called by an admin to get user restrictions set by themselves with `addUserRestriction(ComponentName,String)`.

The target user may have more restrictions set by the system or other admin. To get all the user restrictions currently set, use `UserManager.getUserRestrictions()`.

The profile owner of an organization-owned managed profile may invoke this method on the `DevicePolicyManager` instance it obtained from `getParentProfileInstance(ComponentName)`, for retrieving device-wide restrictions it previously set with `addUserRestriction(ComponentName,String)`.

For callers targeting Android `Build.VERSION_CODES.UPSIDE_DOWN_CAKE` or above, this API will return the local restrictions set on the calling user, or on the parent profile if called from the `DevicePolicyManager` instance obtained from `getParentProfileInstance(ComponentName)`. To get global restrictions set by admin, call `getUserRestrictionsGlobally()` instead.

Note that this is different that the returned restrictions for callers targeting pre Android `Build.VERSION_CODES.UPSIDE_DOWN_CAKE`, were this API returns all local/global restrictions set by the admin on the calling user using `addUserRestriction(ComponentName,String)` or the parent user if called on the `DevicePolicyManager` instance it obtained from `getParentProfileInstance(ComponentName)`.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null`.
Returns	
`Bundle`	a `Bundle` whose keys are the user restrictions, and the values a `boolean` indicating whether the restriction is set. This value cannot be `null`.
Throws	
`SecurityException`	if `admin` is not a device or profile owner.

getUserRestrictionsGlobally

```
public Bundle getUserRestrictionsGlobally ()
```

Called by a profile or device owner to get global user restrictions set with `addUserRestrictionGlobally(String)`.

To get all the user restrictions currently set for a certain user, use `UserManager.getUserRestrictions()`.

Returns	
`Bundle`	a `Bundle` whose keys are the user restrictions, and the values a `boolean` indicating whether the restriction is set. This value cannot be `null`.
Throws	
`IllegalStateException`	if caller is not targeting Android `Build.VERSION_CODES.UPSIDE_DOWN_CAKE` or above.
`SecurityException`	if `admin` is not a device or profile owner.

getWifiMacAddress

```
public String getWifiMacAddress (ComponentName admin)
```

Called by a device owner or profile owner on organization-owned device to get the MAC address of the Wi-Fi device. NOTE: The MAC address returned here should only be used for inventory management and is not likely to be the MAC address used by the device to connect to Wi-Fi networks: MAC addresses used for scanning and connecting to Wi-Fi

networks are randomized by default. To get the randomized MAC address used, call `WifiConfiguration.getRandomizedMacAddress()`.

Parameters	
`admin`	`ComponentName`: Which admin this request is associated with. Null if the caller is not a device admin. This value may be `null`.
Returns	
`String`	the MAC address of the Wi-Fi device, or null when the information is not available. (For example, Wi-Fi hasn't been enabled, or the device doesn't support Wi-Fi.) The address will be in the `XX:XX:XX:XX:XX:XX` format.
Throws	
`SecurityException`	if `admin` is not permitted to get wifi mac addresses

getWifiSsidPolicy

```
public WifiSsidPolicy getWifiSsidPolicy ()
```

Returns the current Wi-Fi SSID policy. If the policy has not been set, it will return NULL.

Returns	
`WifiSsidPolicy`	This value may be `null`.
Throws	
`SecurityException`	if the caller is not a device owner or a profile owner on an organization-owned managed profile.

grantKeyPairToApp

```
public boolean grantKeyPairToApp (ComponentName admin,
                String alias,
                String packageName)
```

Called by a device or profile owner, or delegated certificate chooser (an app that has been delegated the `DELEGATION_CERT_SELECTION` privilege), to grant an application access to an already-installed (or generated) KeyChain key. This is useful (in combination with `installKeyPair(ComponentName, PrivateKey, Certificate, String)` or `generateKeyPair(ComponentName, String, KeyGenParameterSpec, int)`) to let an application call `KeyChain.getPrivateKey(Context, String)` without having to call `KeyChain.choosePrivateKeyAlias(Activity, KeyChainAliasCallback, String, Principal, Uri, String)` first. The grantee app will receive the `KeyChain.ACTION_KEY_ACCESS_CHANGED` broadcast when access to a key is granted. Starting from

`Build.VERSION_CODES.UPSIDE_DOWN_CAKE` throws an `IllegalArgumentException` if `alias` doesn't correspond to an existing key.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with, or `null` if calling from a delegated certificate chooser.
`alias`	`String` : The alias of the key to grant access to. This value cannot be `null` .
`packageName`	`String` : The name of the (already installed) package to grant access to. This value cannot be `null` .

Returns	
`boolean`	`true` if the grant was set successfully, `false` otherwise.

Throws	
`IllegalArgumentException`	if `packageName` or `alias` are empty, or if `packageName` is not a name of an installed package.
`SecurityException`	if the caller is not a device owner, a profile owner or delegated certificate chooser.

grantKeyPairToWifiAuth

```
public boolean grantKeyPairToWifiAuth (String alias)
```

Called by a device or profile owner, or delegated certificate chooser (an app that has been delegated the `DELEGATION_CERT_SELECTION` privilege), to allow using a KeyChain key pair for authentication to Wifi networks. The key can then be used in configurations passed to `WifiManager.addNetwork(WifiConfiguration)` . Starting from `Build.VERSION_CODES.UPSIDE_DOWN_CAKE` throws an `IllegalArgumentException` if `alias` doesn't correspond to an existing key.

Parameters	
`alias`	`String` : The alias of the key pair. This value cannot be `null` .

Returns	
`boolean`	`true` if the operation was set successfully, `false` otherwise.

Throws	
`SecurityException`	if the caller is not a device owner, a profile owner or delegated certificate chooser.

hasCaCertInstalled

```
public boolean hasCaCertInstalled (ComponentName admin,
                byte[] certBuffer)
```

Returns whether this certificate is installed as a trusted CA.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with, or `null` if calling from a delegated certificate installer.
`certBuffer`	`byte` : encoded form of the certificate to look up.
Returns	
`boolean`	
Throws	
`SecurityException`	if `admin` is not `null` and not a device or profile owner.

hasGrantedPolicy

```
public boolean hasGrantedPolicy (ComponentName admin,
                int usesPolicy)
```

Returns true if an administrator has been granted a particular device policy. This can be used to check whether the administrator was activated under an earlier set of policies, but requires additional policies after an upgrade.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. Must be an active administrator, or an exception will be thrown. This value cannot be `null` .
`usesPolicy`	`int` : Which uses-policy to check, as defined in `DeviceAdminInfo` .
Returns	
`boolean`	
Throws	
`SecurityException`	if `admin` is not an active administrator.

hasKeyPair

```
public boolean hasKeyPair (String alias)
```

This API can be called by the following to query whether a certificate and private key are installed under a given alias:

- Device owner
- Profile owner
- Delegated certificate installer
- Credential management app
- An app that holds the `Manifest.permission.MANAGE_DEVICE_POLICY_CERTIFICATES` permission

If called by the credential management app, the alias must exist in the credential management app's `AppUriAuthenticationPolicy`.

Parameters	
`alias`	`String`: The alias under which the key pair is installed. This value cannot be `null`.
Returns	
`boolean`	`true` if a key pair with this alias exists, `false` otherwise.
Throws	
`SecurityException`	if the caller is not a device or profile owner, a delegated certificate installer, the credential management app and does not have the `Manifest.permission.MANAGE_DEVICE_POLICY_CERTIFICATES` permission.

hasLockdownAdminConfiguredNetworks

```
public boolean hasLockdownAdminConfiguredNetworks (ComponentName admin)
```

Called by a device owner or a profile owner of an organization-owned managed profile to determine whether the user is prevented from modifying networks configured by the admin.

Parameters	
`admin`	`ComponentName`: admin Which `DeviceAdminReceiver` this request is associated with. This value may be `null`.
Returns	
`boolean`	
Throws	
`SecurityException`	if caller is not a device owner or a profile owner of an organization-owned managed profile.

installCaCert

```
public boolean installCaCert (ComponentName admin,
                byte[] certBuffer)
```

Installs the given certificate as a user CA.

Inserted user CAs aren't automatically trusted by apps in Android 7.0 (API level 24) and higher. App developers can change the default behavior for an app by adding a Security Configuration File to the app manifest file. The caller must be a profile or device owner on that user, or a delegate package given the `DELEGATION_CERT_INSTALL` scope via `setDelegatedScopes(ComponentName, String, List)`; otherwise a security exception will be thrown.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with, or `null` if calling from a delegated certificate installer.
`certBuffer`	`byte`: encoded form of the certificate to install.
Returns	
`boolean`	false if the certBuffer cannot be parsed or installation is interrupted, true otherwise.
Throws	
`SecurityException`	if `admin` is not `null` and not a device or profile owner.

installExistingPackage

```
public boolean installExistingPackage (ComponentName admin,
                String packageName)
```

Install an existing package that has been installed in another user, or has been kept after removal via `setKeepUninstalledPackages(ComponentName, List)`. This function can be called by a device owner, profile owner or a delegate given the `DELEGATION_INSTALL_EXISTING_PACKAGE` scope via `setDelegatedScopes(ComponentName, String, List)`. When called in a secondary user or managed profile, the user/profile must be affiliated with the device. See `isAffiliatedUser()`.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null`.
`packageName`	`String`: The package to be installed in the calling profile.
Returns	

`boolean`	`true` if the app is installed; `false` otherwise.
Throws	
`SecurityException`	if `admin` is not the device owner, or the profile owner of an affiliated user or profile.

See also:

- `setKeepUninstalledPackages(ComponentName, List)`
- `setDelegatedScopes(ComponentName, String, List)`
- `isAffiliatedUser()`
- `DELEGATION_PACKAGE_ACCESS`

installKeyPair

```
public boolean installKeyPair (ComponentName admin,
                PrivateKey privKey,
                Certificate[] certs,
                String alias,
                int flags)
```

This API can be called by the following to install a certificate chain and corresponding private key for the leaf certificate:

- Device owner
- Profile owner
- Delegated certificate installer
- Credential management app
- An app that holds the `Manifest.permission.MANAGE_DEVICE_POLICY_CERTIFICATES` permission

All apps within the profile will be able to access the certificate chain and use the private key, given direct user approval (if the user is allowed to select the private key).

From Android `Build.VERSION_CODES.S`, the credential management app can call this API. If called by the credential management app:

- The componentName must be `null` r
- The alias must exist in the credential management app's `AppUriAuthenticationPolicy`
- The key pair must not be user selectable

Note, there can only be a credential management app on an unmanaged device.

The caller of this API may grant itself access to the certificate and private key immediately, without user approval. It is a best practice not to request this unless strictly necessary since it opens up additional security vulnerabilities.

Include `INSTALLKEY_SET_USER_SELECTABLE` in the `flags` argument to allow the user to select the key from a dialog.

Note: If the provided `alias` is of an existing alias, all former grants that apps have been given to access the key and certificates associated with this alias will be revoked.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with, or `null` if the caller is not a device admin.
`privKey`	`PrivateKey` : The private key to install. This value cannot be `null` .
`certs`	`Certificate` : The certificate chain to install. The chain should start with the leaf certificate and include the chain of trust in order. This will be returned by `KeyChain.getCertificateChain(Context, String)` . This value cannot be `null` .
`alias`	`String` : The private key alias under which to install the certificate. If a certificate with that alias already exists, it will be overwritten. This value cannot be `null` .
`flags`	`int` : Flags to request that the calling app be granted access to the credentials and set the key to be user-selectable. See `INSTALLKEY_SET_USER_SELECTABLE` and `INSTALLKEY_REQUEST_CREDENTIALS_ACCESS` .

Returns	
`boolean`	`true` if the keys were installed, `false` otherwise.

Throws	
`SecurityException`	if `admin` is not `null` and not a device or profile owner, or `admin` is null but the calling application is not a delegated certificate installer, credential management app and does not have the `Manifest.permission.MANAGE_DEVICE_POLICY_CERTIFICATES` permission.

See also:

- `KeyChain.getCertificateChain(Context, String)`
- `setDelegatedScopes(ComponentName, String, List)`
- `DELEGATION_CERT_INSTALL`

installKeyPair

```
public boolean installKeyPair (ComponentName admin,
                PrivateKey privKey,
                Certificate[] certs,
                String alias,
                boolean requestAccess)
```

This API can be called by the following to install a certificate chain and corresponding private key for the leaf certificate:

- Device owner
- Profile owner
- Delegated certificate installer
- Credential management app
- An app that holds the `Manifest.permission.MANAGE_DEVICE_POLICY_CERTIFICATES` permission

All apps within the profile will be able to access the certificate chain and use the private key, given direct user approval.

From Android `Build.VERSION_CODES.S`, the credential management app can call this API. However, this API sets the key pair as user selectable by default, which is not permitted when called by the credential management app. Instead, `installKeyPair(ComponentName,PrivateKey,Certificate[],String,int)` should be called with `INSTALLKEY_SET_USER_SELECTABLE` not set as a flag. Note, there can only be a credential management app on an unmanaged device.

The caller of this API may grant itself access to the certificate and private key immediately, without user approval. It is a best practice not to request this unless strictly necessary since it opens up additional security vulnerabilities.

Note: If the provided `alias` is of an existing alias, all former grants that apps have been given to access the key and certificates associated with this alias will be revoked.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with, or `null` if the caller is not a device admin.
`privKey`	`PrivateKey`: The private key to install. This value cannot be `null`.
`certs`	`Certificate`: The certificate chain to install. The chain should start with the leaf certificate and include the chain of trust in order. This will be returned by `KeyChain.getCertificateChain(Context, String)`. This value cannot be `null`.
`alias`	`String`: The private key alias under which to install the certificate. If a certificate with that alias already exists, it will be overwritten. This value cannot be `null`.
`requestAccess`	`boolean`: `true` to request that the calling app be granted access to the credentials immediately. Otherwise, access to the credentials will be gated by user approval.
Returns	
`boolean`	`true` if the keys were installed, `false` otherwise.
Throws	

`SecurityException`	if `admin` is not `null` and not a device or profile owner, or `admin` is null and the calling application does not have the `Manifest.permission.MANAGE_DEVICE_POLICY_CERTIFICATES` permission.

See also:

- `KeyChain.getCertificateChain(Context, String)`
- `setDelegatedScopes(ComponentName, String, List)`
- `DELEGATION_CERT_INSTALL`

installKeyPair

```
public boolean installKeyPair (ComponentName admin,
                PrivateKey privKey,
                Certificate cert,
                String alias)
```

This API can be called by the following to install a certificate and corresponding private key:

- Device owner
- Profile owner
- Delegated certificate installer
- Credential management app
- An app that holds the `Manifest.permission.MANAGE_DEVICE_POLICY_CERTIFICATES` permission

All apps within the profile will be able to access the certificate and use the private key, given direct user approval.

From Android `Build.VERSION_CODES.S`, the credential management app can call this API. However, this API sets the key pair as user selectable by default, which is not permitted when called by the credential management app. Instead, `installKeyPair(ComponentName,PrivateKey,Certificate[],String,int)` should be called with `INSTALLKEY_SET_USER_SELECTABLE` not set as a flag.

Access to the installed credentials will not be granted to the caller of this API without direct user approval. This is for security - should a certificate installer become compromised, certificates it had already installed will be protected.

If the installer must have access to the credentials, call `installKeyPair(ComponentName,PrivateKey,Certificate[],String,boolean)` instead.

Note: If the provided `alias` is of an existing alias, all former grants that apps have been given to access the key and certificates associated with this alias will be revoked.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with, or `null` if the caller is not a device admin.

`privKey`	`PrivateKey` : The private key to install. This value cannot be `null` .
`cert`	`Certificate` : The certificate to install. This value cannot be `null` .
`alias`	`String` : The private key alias under which to install the certificate. If a certificate with that alias already exists, it will be overwritten. This value cannot be `null` .

Returns	
`boolean`	`true` if the keys were installed, `false` otherwise.

Throws	
`SecurityException`	if `admin` is not `null` and not a device or profile owner, or `admin` is null and the calling application does not have the `Manifest.permission.MANAGE_DEVICE_POLICY_CERTIFICATES` permission.

installSystemUpdate

```
public void installSystemUpdate (ComponentName admin,
                Uri updateFilePath,
                Executor executor,
                DevicePolicyManager.InstallSystemUpdateCallback callback)
```

Called by device owner or profile owner of an organization-owned managed profile to install a system update from the given file. The device will be rebooted in order to finish installing the update. Note that if the device is rebooted, this doesn't necessarily mean that the update has been applied successfully. The caller should additionally check the system version with `Build.FINGERPRINT` or `Build.VERSION` . If an error occurs during processing the OTA before the reboot, the caller will be notified by `InstallSystemUpdateCallback` . If device does not have sufficient battery level, the installation will fail with error `InstallSystemUpdateCallback.UPDATE_ERROR_BATTERY_LOW` .

Parameters	
`admin`	`ComponentName` : The `DeviceAdminReceiver` that this request is associated with. Null if the caller is not a device admin. This value may be `null` .
`updateFilePath`	`Uri` : A Uri of the file that contains the update. The file should be readable by the calling app. This value cannot be `null` .
`executor`	`Executor` : The executor through which the callback should be invoked. This value cannot be `null` . Callback and listener events are dispatched through this `Executor` , providing an easy way to control which thread is used. To dispatch events through the main thread of your application, you

	can use `Context.getMainExecutor()`. Otherwise, provide an `Executor` that dispatches to an appropriate thread.
`callback`	`DevicePolicyManager.InstallSystemUpdateCallback`: A callback object that will inform the caller when installing an update fails. This value cannot be `null`.

isActivePasswordSufficient

```
public boolean isActivePasswordSufficient ()
```

Determines whether the calling user's current password meets policy requirements (e.g. quality, minimum length). The user must be unlocked to perform this check.

Policy requirements which affect this check can be set by admins of the user, but also by the admin of a managed profile associated with the calling user (when the managed profile doesn't have a separate work challenge). When a managed profile has a separate work challenge, its policy requirements only affect the managed profile.

Depending on the user, this method checks the policy requirement against one of the following passwords:

- For the primary user or secondary users: the personal keyguard password.
- For managed profiles: a work challenge if set, otherwise the parent user's personal keyguard password.

  In other words, it's always checking the requirement against the password that is protecting the calling user.

  Note that this method considers all policy requirements targeting the password in question. For example a profile owner might set a requirement on the parent profile i.e. personal keyguard but not on the profile itself. When the device has a weak personal keyguard password and no separate work challenge, calling this method will return `false` despite the profile owner not setting a policy on the profile itself. This is because the profile's current password is the personal keyguard password, and it does not meet all policy requirements.

  Device admins must request `DeviceAdminInfo.USES_POLICY_LIMIT_PASSWORD` before calling this method. Note, this policy type is deprecated for device admins in Android 9.0 (API level 28) or higher.

  This method can be called on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to determine if the password set on the parent profile is sufficient.

  On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, the password is always treated as empty - i.e. this method will always return false on such devices, provided any password requirements were set.

Returns	
`boolean`	`true` if the password meets the policy requirements, `false` otherwise

Throws	
`IllegalStateException`	if the user isn't unlocked
`SecurityException`	if the calling application isn't an active admin that uses `DeviceAdminInfo.USES_POLICY_LIMIT_PASSWORD`

isActivePasswordSufficientForDeviceRequirement

```
public boolean isActivePasswordSufficientForDeviceRequirement ()
```

Called by profile owner of a managed profile to determine whether the current device password meets policy requirements set explicitly device-wide.

This API is similar to `isActivePasswordSufficient()`, with two notable differences:

- this API always targets the device password. As a result it should always be called on the `getParentProfileInstance(ComponentName)` instance.
- password policy requirement set on the managed profile is not taken into consideration by this API, even if the device currently does not have a separate work challenge set.

This API is designed to facilite progressive password enrollment flows when the DPC imposes both device and profile password policies. DPC applies profile password policy by calling `setPasswordQuality(ComponentName,int)` or `setRequiredPasswordComplexity(int)` on the regular `DevicePolicyManager` instance, while it applies device-wide policy by calling `setRequiredPasswordComplexity(int)` on the `getParentProfileInstance(ComponentName)` instance. The DPC can utilize this check to guide the user to set a device password first taking into consideration the device-wide policy only, and then prompt the user to either upgrade it to be fully compliant, or enroll a separate work challenge to satisfy the profile password policy only.

The device user must be unlocked (@link `UserManager.isUserUnlocked(UserHandle)`) to perform this check.

Returns	
`boolean`	`true` if the device password meets explicit requirement set on it, `false` otherwise.
Throws	
`IllegalStateException`	if the user isn't unlocked
`SecurityException`	if the calling application is not a profile owner of a managed profile, or if this API is not called on the parent DevicePolicyManager instance.

isAdminActive

```
public boolean isAdminActive (ComponentName admin)
```

Return true if the given administrator component is currently active (enabled) in the system.

Parameters	
`admin`	`ComponentName` : The administrator component to check for. This value cannot be `null` .
Returns	
`boolean`	`true` if `admin` is currently enabled in the system, `false` otherwise

isAffiliatedUser

```
public boolean isAffiliatedUser ()
```

Returns whether this user is affiliated with the device.

By definition, the user that the device owner runs on is always affiliated with the device. Any other user is considered affiliated with the device if the set specified by its profile owner via `setAffiliationIds(ComponentName, Set)` intersects with the device owner's.

Returns	
`boolean`	

isAlwaysOnVpnLockdownEnabled

```
public boolean isAlwaysOnVpnLockdownEnabled (ComponentName admin)
```

Called by device or profile owner to query whether current always-on VPN is configured in lockdown mode. Returns `false` when no always-on configuration is set.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
Returns	
`boolean`	
Throws	
`SecurityException`	if `admin` is not a device or a profile owner.

isApplicationHidden

```
public boolean isApplicationHidden (ComponentName admin,
                String packageName)
```

Determine if a package is hidden. This function can be called by a device owner, profile owner, or by a delegate given the `DELEGATION_PACKAGE_ACCESS` scope via `setDelegatedScopes(ComponentName, String, List)`.

This method can be called on the `DevicePolicyManager` instance, returned by `getParentProfileInstance(ComponentName)`, where the caller must be the profile owner of an organization-owned managed profile and the package must be a system package. If called on the parent instance, this will determine whether the package is hidden or unhidden in the personal profile.

Starting from `Build.VERSION_CODES.UPSIDE_DOWN_CAKE`, the returned policy will be the current resolved policy rather than the policy set by the calling admin.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with, or `null` if the caller is not a device admin.
`packageName`	`String`: The name of the package to retrieve the hidden status of.
Returns	
`boolean`	boolean `true` if the package is hidden, `false` otherwise.
Throws	
`IllegalArgumentException`	if called on the parent profile and the package provided is not a system package.
`SecurityException`	if `admin` is not a device or profile owner or if called on the parent profile and the `admin` is not a profile owner of an organization-owned managed profile.

isBackupServiceEnabled

```
public boolean isBackupServiceEnabled (ComponentName admin)
```

Return whether the backup service is enabled by the device owner or profile owner for the current user, as previously set by `setBackupServiceEnabled(ComponentName,boolean)`.

Whether the backup functionality is actually enabled or not depends on settings from both the current user and the device owner, please see `setBackupServiceEnabled(ComponentName,boolean)` for details.

Backup service manages all backup and restore mechanisms on the device.

Parameters	
`admin`	`ComponentName` : This value cannot be `null` .

Returns	
`boolean`	`true` if backup service is enabled, `false` otherwise.

isCallerApplicationRestrictionsManagingPackage

```
public boolean isCallerApplicationRestrictionsManagingPackage ()
```

This method was deprecated in API level 26.
From `Build.VERSION_CODES.O` . Use `getDelegatedScopes(ComponentName, String)` instead.

Called by any application to find out whether it has been granted permission via `setApplicationRestrictionsManagingPackage(ComponentName, String)` to manage application restrictions for the calling user.

This is done by comparing the calling Linux uid with the uid of the package specified by that method.

Returns	
`boolean`	

isCommonCriteriaModeEnabled

```
public boolean isCommonCriteriaModeEnabled (ComponentName admin)
```

Returns whether Common Criteria mode is currently enabled. Device owner and profile owner of an organization-owned managed profile can query its own Common Criteria mode setting by calling this method with its admin `ComponentName` . Any caller can obtain the aggregated device-wide Common Criteria mode state by passing `null` as the `admin` argument.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null` .

Returns	
`boolean`	`true` if Common Criteria mode is enabled, `false` otherwise.

isComplianceAcknowledgementRequired

```
public boolean isComplianceAcknowledgementRequired ()
```

Called by a profile owner of an organization-owned managed profile to query whether it needs to acknowledge device compliance to allow the user to turn the profile off if needed according to the maximum profile time off policy. Normally when acknowledgement is needed the DPC gets a `DeviceAdminReceiver.onComplianceAcknowledgementRequired(Context,Intent)` callback. But if the callback was not delivered or handled for some reason, this method can be used to verify if acknowledgement is needed.

Returns	
`boolean`	
Throws	
`IllegalStateException`	if the user isn't unlocked

See also:

- `acknowledgeDeviceCompliant()`
- `setManagedProfileMaximumTimeOff(ComponentName,long)`
- `DeviceAdminReceiver.onComplianceAcknowledgementRequired(Context,Intent)`

isDeviceFinanced

```
public boolean isDeviceFinanced ()
```

Returns `true` if this device is marked as a financed device.

A financed device can be entered into lock task mode (see `setLockTaskPackages(ComponentName, String)`) by the holder of the role `android.app.role.RoleManager#ROLE_FINANCED_DEVICE_KIOSK` . If this occurs, Device Owners and Profile Owners that have set lock task packages or features, or that attempt to set lock task packages or features, will receive a callback indicating that it could not be set. See `PolicyUpdateReceiver.onPolicyChanged` and `PolicyUpdateReceiver.onPolicySetResult` .

To be informed of changes to this status you can subscribe to the broadcast `ACTION_DEVICE_FINANCING_STATE_CHANGED` .

Returns	
`boolean`	
Throws	
`SecurityException`	if the caller is not a device owner, profile owner of an organization-owned managed profile, profile owner on the primary user or holder of one of the following roles: `android.app.role.RoleManager.ROLE_DEVICE_POLICY_MANAGEMENT` , `android.app.role.RoleManager.ROLE_SYSTEM_SUPERVISION` .

isDeviceIdAttestationSupported

```
public boolean isDeviceIdAttestationSupported ()
```

Returns `true` if the device supports attestation of device identifiers in addition to key attestation. See `generateKeyPair(ComponentName,String,KeyGenParameterSpec,int)`

Returns	
`boolean`	`true` if Device ID attestation is supported.

isDeviceOwnerApp

```
public boolean isDeviceOwnerApp (String packageName)
```

Used to determine if a particular package has been registered as a Device Owner app. A device owner app is a special device admin that cannot be deactivated by the user, once activated as a device admin. It also cannot be uninstalled. To check whether a particular package is currently registered as the device owner app, pass in the package name from `Context.getPackageName()` to this method.

This is useful for device admin apps that want to check whether they are also registered as the device owner app. The exact mechanism by which a device admin app is registered as a device owner app is defined by the setup process.

Parameters	
`packageName`	`String` : the package name of the app, to compare with the registered device owner app, if any.
Returns	
`boolean`	whether or not the package is registered as the device owner app.

isEphemeralUser

```
public boolean isEphemeralUser (ComponentName admin)
```

Checks if the profile owner is running in an ephemeral user.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
Returns	
`boolean`	whether the profile owner is running in an ephemeral user.

isKeyPairGrantedToWifiAuth

```
public boolean isKeyPairGrantedToWifiAuth (String alias)
```

Called by a device or profile owner, or delegated certificate chooser (an app that has been delegated the `DELEGATION_CERT_SELECTION` privilege), to query whether a KeyChain key pair can be used for authentication to Wifi networks.

Parameters	
`alias`	`String` : The alias of the key pair. This value cannot be `null` .
Returns	
`boolean`	`true` if the key pair can be used, `false` otherwise.
Throws	
`IllegalArgumentException`	if `alias` does not correspond to an existing key
`SecurityException`	if the caller is not a device owner, a profile owner or delegated certificate chooser.

isLockTaskPermitted

```
public boolean isLockTaskPermitted (String pkg)
```

This function lets the caller know whether the given component is allowed to start the lock task mode.

Parameters	
`pkg`	`String` : The package to check
Returns	
`boolean`	

isLogoutEnabled

```
public boolean isLogoutEnabled ()
```

Returns whether logout is enabled by a device owner.

Returns	
`boolean`	`true` if logout is enabled by device owner, `false` otherwise.

isManagedProfile

```
public boolean isManagedProfile (ComponentName admin)
```

Return if this user is a managed profile of another user. An admin can become the profile owner of a managed profile with `ACTION_PROVISION_MANAGED_PROFILE` and of a managed user with `createAndManageUser(ComponentName, String, ComponentName, PersistableBundle, int)`

Parameters	
`admin`	`ComponentName` : Which profile owner this request is associated with. This value cannot be `null` .
Returns	
`boolean`	if this user is a managed profile of another user.

isMasterVolumeMuted

```
public boolean isMasterVolumeMuted (ComponentName admin)
```

Called by profile or device owners to check whether the global volume mute is on or off.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
Returns	
`boolean`	`true` if global volume is muted, `false` if it's not.
Throws	
`SecurityException`	if `admin` is not a device or profile owner.

isMtePolicyEnforced

```
public static boolean isMtePolicyEnforced ()
```

Get the current MTE state of the device. Learn more about MTE

Returns	
`boolean`	whether MTE is currently enabled on the device.

isNetworkLoggingEnabled

```
public boolean isNetworkLoggingEnabled (ComponentName admin)
```

Return whether network logging is enabled by a device owner or profile owner of a managed profile.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. Can only be `null` if the caller is a delegated app with `DELEGATION_NETWORK_LOGGING` or has MANAGE_USERS permission.
Returns	
`boolean`	`true` if network logging is enabled by device owner or profile owner, `false` otherwise.
Throws	
`SecurityException`	if `admin` is not a device owner or profile owner and caller has no MANAGE_USERS permission

isOrganizationOwnedDeviceWithManagedProfile

```
public boolean isOrganizationOwnedDeviceWithManagedProfile ()
```

Apps can use this method to find out if the device was provisioned as organization-owend device with a managed profile. This, together with checking whether the device has a device owner (by calling `isDeviceOwnerApp(String)`), could be used to learn whether the device is owned by an organization or an individual: If this method returns true OR `isDeviceOwnerApp(String)` returns true (for any package), then the device is owned by an organization. Otherwise, it's owned by an individual.

Returns	
`boolean`	`true` if the device was provisioned as organization-owned device, `false` otherwise.

isOverrideApnEnabled

```
public boolean isOverrideApnEnabled (ComponentName admin)
```

Called by device owner to check if override APNs are currently enabled.

Parameters	
`admin`	`ComponentName` : which `DeviceAdminReceiver` this request is associated with. This value cannot be `null`.
Returns	
`boolean`	`true` if override APNs are currently enabled, `false` otherwise.

Throws	
SecurityException	if admin is not a device owner.

isPackageSuspended

```
public boolean isPackageSuspended (ComponentName admin,
                String packageName)
```

Determine if a package is suspended. This function can be called by a device owner, profile owner, or by a delegate given the DELEGATION_PACKAGE_ACCESS scope via setDelegatedScopes(ComponentName, String, List) or by holders of the permission Manifest.permission.MANAGE_DEVICE_POLICY_PACKAGE_STATE .

Parameters	
admin	ComponentName : Which DeviceAdminReceiver this request is associated with. Null if the caller is not a device admin. This value may be null .
packageName	String : The name of the package to retrieve the suspended status of.
Returns	
boolean	true if the package is suspended or false if the package is not suspended, could not be found or an error occurred.
Throws	
PackageManager.NameNotFoundException	if the package could not be found.
SecurityException	if admin is not a device or profile owner or has not been granted the permission Manifest.permission.MANAGE_DEVICE_POLICY_PACKAGE_STATE .

isPreferentialNetworkServiceEnabled

```
public boolean isPreferentialNetworkServiceEnabled ()
```

Indicates whether preferential network service is enabled.

Before Android version Build.VERSION_CODES.TIRAMISU : This method can be called by the profile owner of a managed profile.

Starting from Android version Build.VERSION_CODES.TIRAMISU : This method can be called by the profile owner of a managed profile or device owner.

Returns	
`boolean`	whether preferential network service is enabled.
Throws	
`SecurityException`	if the caller is not the profile owner or device owner.

isProfileOwnerApp

```
public boolean isProfileOwnerApp (String packageName)
```

Used to determine if a particular package is registered as the profile owner for the user. A profile owner is a special device admin that has additional privileges within the profile.

Parameters	
`packageName`	`String` : The package name of the app to compare with the registered profile owner.
Returns	
`boolean`	Whether or not the package is registered as the profile owner.

isProvisioningAllowed

```
public boolean isProvisioningAllowed (String action)
```

Returns whether it is possible for the caller to initiate provisioning of a managed profile or device, setting itself as the device or profile owner.

Parameters	
`action`	`String` : One of `ACTION_PROVISION_MANAGED_DEVICE` , `ACTION_PROVISION_MANAGED_PROFILE` , `ERROR(/#ACTION_PROVISION_MULTIUSER_MANAGED_USER)` . This value cannot be `null` .
Returns	
`boolean`	whether provisioning a managed profile or device is possible.
Throws	
`IllegalArgumentException`	if the supplied action is not valid.

isResetPasswordTokenActive

```
public boolean isResetPasswordTokenActive (ComponentName admin)
```

Called by a profile, device owner or a holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_RESET_PASSWORD` to check if the current reset password token is active.

On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, false is always returned.
Requires the `PackageManager#FEATURE_SECURE_LOCK_SCREEN` feature which can be detected using `PackageManager.hasSystemFeature(String)` .

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null` .

Returns	
`boolean`	true if the token is active, false otherwise.

Throws	
`IllegalStateException`	if no token has been set.
`SecurityException`	if admin is not a device or profile owner and not a holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_RESET_PASSWORD`

isSafeOperation

```
public boolean isSafeOperation (int reason)
```

Checks if it's safe to run operations that can be affected by the given `reason` .

Note: notice that the operation safety state might change between the time this method returns and the operation's method is called, so calls to the latter could still throw a `UnsafeStateException` even when this method returns `true` .

Parameters	
`reason`	`int` : currently, only supported reason is `OPERATION_SAFETY_REASON_DRIVING_DISTRACTION` . Value is one of the following: • `OPERATION_SAFETY_REASON_DRIVING_DISTRACTION`

Returns	
`boolean`	whether it's safe to run operations that can be affected by the given `reason` .

isSecurityLoggingEnabled

```
public boolean isSecurityLoggingEnabled (ComponentName admin)
```

Return whether security logging is enabled or not by the admin.

Can only be called by the device owner or a profile owner of an organization-owned managed profile, otherwise a `SecurityException` will be thrown.

Parameters	
`admin`	`ComponentName` : Which device admin this request is associated with. Null if the caller is not a device admin. This value may be `null` .
Returns	
`boolean`	`true` if security logging is enabled, `false` otherwise.
Throws	
`SecurityException`	if the caller is not allowed to control security logging.

isStatusBarDisabled

```
public boolean isStatusBarDisabled ()
```

Returns whether the status bar is disabled/enabled, see `setStatusBarDisabled(ComponentName, boolean)` .

Callable by device owner or profile owner of secondary users that is affiliated with the device owner.

This policy has no effect in LockTask mode. The behavior of the status bar in LockTask mode can be configured with `setLockTaskFeatures(ComponentName,int)` .

This policy also does not have any effect while on the lock screen, where the status bar will not be disabled.

Returns	
`boolean`	
Throws	
`SecurityException`	if the caller is not the device owner, or a profile owner of secondary user that is affiliated with the device.

isUninstallBlocked

```
public boolean isUninstallBlocked (ComponentName admin,
                String packageName)
```

Check whether the user has been blocked by device policy from uninstalling a package. Requires the caller to be the profile owner if checking a specific admin's policy.

Note: Starting from `Build.VERSION_CODES.LOLLIPOP_MR1`, the behavior of this API is changed such that passing `null` as the `admin` parameter will return if any admin has blocked the uninstallation. Before L MR1, passing `null` will cause a NullPointerException to be raised.

Note: If your app targets Android 11 (API level 30) or higher, this method returns a filtered result. Learn more about how to manage package visibility.

Starting from `Build.VERSION_CODES.UPSIDE_DOWN_CAKE`, the returned policy will be the current resolved policy rather than the policy set by the calling admin.

Parameters	
`admin`	`ComponentName`: The name of the admin component whose blocking policy will be checked, or `null` to check whether any admin has blocked the uninstallation. Starting from `Build.VERSION_CODES.UPSIDE_DOWN_CAKE` admin will be ignored and assumed `null`.
`packageName`	`String`: package to check.
Returns	
`boolean`	true if uninstallation is blocked and the given package is visible to you, false otherwise if uninstallation isn't blocked or the given package isn't visible to you.
Throws	
`SecurityException`	if `admin` is not a device or profile owner.

isUniqueDeviceAttestationSupported

```
public boolean isUniqueDeviceAttestationSupported ()
```

Returns `true` if the StrongBox Keymaster implementation on the device was provisioned with an individual attestation certificate and can sign attestation records using it (as attestation using an individual attestation certificate is a feature only Keymaster implementations with StrongBox security level can implement). For use prior to calling `generateKeyPair(ComponentName,String,KeyGenParameterSpec,int)`.

Returns	
`boolean`	`true` if individual attestation is supported.

isUsbDataSignalingEnabled

```
public boolean isUsbDataSignalingEnabled ()
```

Returns whether USB data signaling is currently enabled.

When called by a device owner or profile owner of an organization-owned managed profile, this API returns whether USB data signaling is currently enabled by that admin. When called by any other app, returns whether USB data signaling is currently enabled on the device.

Returns	
`boolean`	`true` if USB data signaling is enabled, `false` otherwise.

isUsingUnifiedPassword

```
public boolean isUsingUnifiedPassword (ComponentName admin)
```

When called by a profile owner of a managed profile returns true if the profile uses unified challenge with its parent user. **Note**: This method is not concerned with password quality and will return false if the profile has empty password as a separate challenge.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null`.
Returns	
`boolean`	
Throws	
`SecurityException`	if `admin` is not a profile owner of a managed profile.

listForegroundAffiliatedUsers

```
public List<UserHandle> listForegroundAffiliatedUsers ()
```

Gets the list of `affiliated` users running on foreground.

Returns	
`List<UserHandle>`	list of `affiliated` users running on foreground. This value cannot be `null`.
Throws	

`SecurityException`	if the calling application is not a device owner

lockNow

```
public void lockNow ()
```

Make the device lock immediately, as if the lock screen timeout has expired at the point of this call.

This method secures the device in response to an urgent situation, such as a lost or stolen device. After this method is called, the device must be unlocked using strong authentication (PIN, pattern, or password). This API is intended for use only by device admins.

From version `Build.VERSION_CODES.R` onwards, the caller must either have the LOCK_DEVICE permission or the device must have the device admin feature; if neither is true, then the method will return without completing any action. Before version `Build.VERSION_CODES.R`, the device needed the device admin feature, regardless of the caller's permissions.

The calling device admin must have requested `DeviceAdminInfo.USES_POLICY_FORCE_LOCK` to be able to call this method; if it has not, a security exception will be thrown.

If there's no lock type set, this method forces the device to go to sleep but doesn't lock the device. Device admins who find the device in this state can lock an otherwise-insecure device by first calling `resetPassword(String, int)` to set the password and then lock the device.

This method can be called on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to lock the parent profile.

NOTE: on `automotive builds`, this method doesn't turn off the screen as it would be a driving safety distraction.

Equivalent to calling `lockNow(int)` with no flags.

Throws	
`SecurityException`	if the calling application does not own an active administrator that uses `DeviceAdminInfo.USES_POLICY_FORCE_LOCK`

lockNow

```
public void lockNow (int flags)
```

Make the device lock immediately, as if the lock screen timeout has expired at the point of this call.

This method secures the device in response to an urgent situation, such as a lost or stolen device. After this method is called, the device must be unlocked using strong authentication (PIN, pattern, or password). This API is intended for use only by device admins.

From version `Build.VERSION_CODES.R` onwards, the caller must either have the LOCK_DEVICE permission or the device must have the device admin feature; if neither is true, then the method will return without completing any action. Before version `Build.VERSION_CODES.R`, the device needed the device admin feature, regardless of the caller's permissions.

A calling device admin must have requested `DeviceAdminInfo.USES_POLICY_FORCE_LOCK` to be able to call this method; if it has not, a security exception will be thrown.

If there's no lock type set, this method forces the device to go to sleep but doesn't lock the device. Device admins who find the device in this state can lock an otherwise-insecure device by first calling `resetPassword(String, int)` to set the password and then lock the device.

This method can be called on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to lock the parent profile as well as the managed profile.

NOTE: In order to lock the parent profile and evict the encryption key of the managed profile, `lockNow()` must be called twice: First, `lockNow()` should be called on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)`, then `lockNow(int)` should be called on the `DevicePolicyManager` instance associated with the managed profile, with the `FLAG_EVICT_CREDENTIAL_ENCRYPTION_KEY` flag. Calling the method twice in this order ensures that all users are locked and does not stop the device admin on the managed profile from issuing a second call to lock its own profile.

NOTE: on `automotive builds`, this method doesn't turn off the screen as it would be a driving safety distraction.

Parameters	
`flags`	`int`: May be 0 or `FLAG_EVICT_CREDENTIAL_ENCRYPTION_KEY`. Value is either `0` or • `FLAG_EVICT_CREDENTIAL_ENCRYPTION_KEY`

Throws	
`IllegalArgumentException`	if the `FLAG_EVICT_CREDENTIAL_ENCRYPTION_KEY` flag is passed when locking the parent profile.
`SecurityException`	if the calling application does not own an active administrator that uses `DeviceAdminInfo.USES_POLICY_FORCE_LOCK` and the does not hold the LOCK_DEVICE permission, or the `FLAG_EVICT_CREDENTIAL_ENCRYPTION_KEY` flag is passed by an application that is not a profile owner of a managed profile.
`UnsupportedOperationException`	if the `FLAG_EVICT_CREDENTIAL_ENCRYPTION_KEY` flag is passed when `getStorageEncryptionStatus()` does not return `ENCRYPTION_STATUS_ACTIVE_PER_USER`.

logoutUser

```
public int logoutUser (ComponentName admin)
```

Called by a profile owner of secondary user that is affiliated with the device to stop the calling user and switch back to primary user (when the user was `switchUser(ComponentName,UserHandle)` switched to) or stop the user (when it was `started in background` .

Notice that on devices running with `headless system user mode` , there is no primary user, so it switches back to the user that was in the foreground before the first call to `switchUser(ComponentName,UserHandle)` (or fails with `UserManager.USER_OPERATION_ERROR_UNKNOWN` if that method was not called prior to this call).

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .

Returns	
`int`	one of the following result codes: `UserManager.USER_OPERATION_ERROR_UNKNOWN` , `UserManager.USER_OPERATION_SUCCESS` , `UserManager.USER_OPERATION_ERROR_MANAGED_PROFILE` , `UserManager.USER_OPERATION_ERROR_CURRENT_USER` Value is one of the following: • `UserManager.USER_OPERATION_SUCCESS` • `UserManager.USER_OPERATION_ERROR_UNKNOWN` • `UserManager.USER_OPERATION_ERROR_MANAGED_PROFILE` • `UserManager.USER_OPERATION_ERROR_MAX_RUNNING_USERS` • `UserManager.USER_OPERATION_ERROR_CURRENT_USER` • `UserManager.USER_OPERATION_ERROR_LOW_STORAGE` • `UserManager.USER_OPERATION_ERROR_MAX_USERS`

Throws	
`SecurityException`	if `admin` is not a profile owner affiliated with the device.

reboot

```
public void reboot (ComponentName admin)
```

Called by device owner to reboot the device. If there is an ongoing call on the device, throws an `IllegalStateException` .

Parameters	
`admin`	`ComponentName` : Which device owner the request is associated with. This value cannot be `null` .

Throws	
`IllegalStateException`	if device has an ongoing call.
`SecurityException`	if `admin` is not a device owner.

removeActiveAdmin

```
public void removeActiveAdmin (ComponentName admin)
```

Remove a current administration component. This can only be called by the application that owns the administration component; if you try to remove someone else's component, a security exception will be thrown.

Note that the operation is not synchronous and the admin might still be active (as indicated by `getActiveAdmins()`) by the time this method returns.

Parameters	
`admin`	`ComponentName`: The administration compononent to remove. This value cannot be `null`.
Throws	
`SecurityException`	if the caller is not in the owner application of `admin`.

removeCrossProfileWidgetProvider

```
public boolean removeCrossProfileWidgetProvider (ComponentName admin,
                String packageName)
```

This method was deprecated in API level 37.
While this API still works to mutate the current allowlist, please consider switching to `setCrossProfileWidgetProviders(Set)` for better performance.

Called by the profile owner of a managed profile or a holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_PROFILE_INTERACTION` to disable widget providers from a given package to be available in the parent profile. For this method to take effect the package should have been added via `addCrossProfileWidgetProvider(android.content.ComponentName,String)`.

Note: By default no widget provider package is allowlisted.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null`.

`packageName`	`String` : The package from which widget providers are no longer allowlisted.

Returns	
`boolean`	Whether the package was removed.

Throws	
`SecurityException`	if `admin` is not a profile owner and not a holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_PROFILE_INTERACTION` .

See also:

- `addCrossProfileWidgetProvider(android.content.ComponentName,String)`
- `getCrossProfileWidgetProviders(android.content.ComponentName)`

removeKeyPair

```
public boolean removeKeyPair (ComponentName admin,
                String alias)
```

This API can be called by the following to remove a certificate and private key pair installed under a given alias:

- Device owner
- Profile owner
- Delegated certificate installer
- Credential management app

From Android `Build.VERSION_CODES.S` , the credential management app can call this API. If called by the credential management app, the componentName must be `null` . Note, there can only be a credential management app on an unmanaged device.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with, or `null` if the caller is not a device admin.
`alias`	`String` : The private key alias under which the certificate is installed. This value cannot be `null` .

Returns	
`boolean`	`true` if the private key alias no longer exists, `false` otherwise.

Throws	
`SecurityException`	if `admin` is not `null` and not a device or profile owner, or `admin` is null but the calling application is not a delegated certificate installer or credential management app.

removeOverrideApn

```
public boolean removeOverrideApn (ComponentName admin,
                int apnId)
```

Called by device owner or managed profile owner to remove an override APN.

This method may returns `false` if there is no override APN with the given `apnId`.

Before Android version `Build.VERSION_CODES.TIRAMISU` : Only device owners can remove APNs.

Starting from Android version `Build.VERSION_CODES.TIRAMISU` : Both device owners and managed profile owners can remove enterprise APNs (`ApnSetting.TYPE_ENTERPRISE`), while only device owners can remove other type of APNs.

Parameters	
`admin`	`ComponentName` : which `DeviceAdminReceiver` this request is associated with. This value cannot be `null`.
`apnId`	`int` : the `id` of the override APN to remove
Returns	
`boolean`	`true` if the required override APN is successfully removed, `false` otherwise.
Throws	
`SecurityException`	If request is for enterprise APN `admin` is either device owner or profile owner and in all other types of APN if `admin` is not a device owner.

removeUser

```
public boolean removeUser (ComponentName admin,
                UserHandle userHandle)
```

Called by a device owner to remove a user/profile and all associated data. The primary user can not be removed.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null`.
`userHandle`	`UserHandle` : the user to remove. This value cannot be `null`.
Returns	
`boolean`	`true` if the user was removed, `false` otherwise.

Throws	
SecurityException	if admin is not a device owner.

requestBugreport

```
public boolean requestBugreport (ComponentName admin)
```

Called by a device owner to request a bugreport.

If the device contains secondary users or profiles, they must be affiliated with the device. Otherwise a SecurityException will be thrown. See isAffiliatedUser() .

Parameters	
admin	ComponentName : Which DeviceAdminReceiver this request is associated with. This value cannot be null .
Returns	
boolean	true if the bugreport collection started successfully, or false if it wasn't triggered because a previous bugreport operation is still active (either the bugreport is still running or waiting for the user to share or decline)
Throws	
SecurityException	if admin is not a device owner, or there is at least one profile or secondary user that is not affiliated with the device.

resetPassword

```
public boolean resetPassword (String password,
                int flags)
```

This method was deprecated in API level 30.
Please use resetPasswordWithToken(ComponentName, String, byte, int) instead.

Force a new password for device unlock (the password needed to access the entire device) or the work profile challenge on the current user. This takes effect immediately.

Before Build.VERSION_CODES.N , this API is available to device admin, profile owner and device owner. Starting from Build.VERSION_CODES.N , legacy device admin (who is not also profile owner or device owner) can only call this API to set a new password if there is currently no password set. Profile owner and device owner can continue to force change an existing password as long as the target user is unlocked, although device owner will not be able to call this API at all if there is also a managed profile on the device.

Between `Build.VERSION_CODES.O` , `Build.VERSION_CODES.P` and `Build.VERSION_CODES.Q` , profile owner and devices owner targeting SDK level `Build.VERSION_CODES.O` or above who attempt to call this API will receive `SecurityException` ; they are encouraged to migrate to the new `resetPasswordWithToken(ComponentName, String, byte, int)` API instead. Profile owner and device owner targeting older SDK levels are not affected: they continue to experience the existing behaviour described in the previous paragraph.

Starting from `Build.VERSION_CODES.R` , this API is no longer supported in most cases. Device owner and profile owner calling this API will receive `SecurityException` if they target SDK level `Build.VERSION_CODES.O` or above, or they will receive a silent failure (API returning `false`) if they target lower SDK level. For legacy device admins, this API throws `SecurityException` if they target SDK level `Build.VERSION_CODES.N` or above, and returns `false` otherwise. Only privileged apps holding RESET_PASSWORD permission which are part of the system factory image can still call this API to set a new password if there is currently no password set. In this case, if the device already has a password, this API will throw `SecurityException` .

The given password must be sufficient for the current password quality and length constraints as returned by `getPasswordQuality(ComponentName)` and `getPasswordMinimumLength(ComponentName)` ; if it does not meet these constraints, then it will be rejected and false returned. Note that the password may be a stronger quality (containing alphanumeric characters when the requested quality is only numeric), in which case the currently active quality will be increased to match.

On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, this methods does nothing.

The calling device admin must have requested `DeviceAdminInfo.USES_POLICY_RESET_PASSWORD` to be able to call this method; if it has not, a security exception will be thrown.
Requires the `PackageManager#FEATURE_SECURE_LOCK_SCREEN` feature which can be detected using `PackageManager.hasSystemFeature(String)` .

Parameters	
`password`	`String` : The new password for the user. Null or empty clears the password.
`flags`	`int` : May be 0 or combination of `RESET_PASSWORD_REQUIRE_ENTRY` and `RESET_PASSWORD_DO_NOT_ASK_CREDENTIALS_ON_BOOT` .
Returns	
`boolean`	Returns true if the password was applied, or false if it is not acceptable for the current constraints.
Throws	
`IllegalStateException`	if the calling user is locked or has a managed profile.
`SecurityException`	if the calling application does not own an active administrator that uses `DeviceAdminInfo.USES_POLICY_RESET_PASSWORD`

resetPasswordWithToken

```
public boolean resetPasswordWithToken (ComponentName admin,
                String password,
                byte[] token,
                int flags)
```

Called by device or profile owner to force set a new device unlock password or a managed profile challenge on current user. This takes effect immediately.

Unlike `resetPassword(String, int)`, this API can change the password even before the user or device is unlocked or decrypted. The supplied token must have been previously provisioned via `setResetPasswordToken(ComponentName, byte)`, and in active state `isResetPasswordTokenActive(ComponentName)`.

The given password must be sufficient for the current password quality and length constraints as returned by `getPasswordQuality(ComponentName)` and `getPasswordMinimumLength(ComponentName)`; if it does not meet these constraints, then it will be rejected and false returned. Note that the password may be a stronger quality, for example, a password containing alphanumeric characters when the requested quality is only numeric.

Calling with a `null` or empty password will clear any existing PIN, pattern or password if the current password constraints allow it.

On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, calling this methods has no effect - the password is always empty - and false is returned.
Requires the `PackageManager#FEATURE_SECURE_LOCK_SCREEN` feature which can be detected using `PackageManager.hasSystemFeature(String)`.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null`.
`password`	`String`: The new password for the user. `null` or empty clears the password.
`token`	`byte`: the password reset token previously provisioned by `setResetPasswordToken(ComponentName, byte)`.
`flags`	`int`: May be 0 or combination of `RESET_PASSWORD_REQUIRE_ENTRY` and `RESET_PASSWORD_DO_NOT_ASK_CREDENTIALS_ON_BOOT`.

Returns	
`boolean`	Returns true if the password was applied, or false if it is not acceptable for the current constraints.

Throws	
`IllegalStateException`	if the provided token is not valid.
`SecurityException`	if admin is not a device or profile owner.

retrieveNetworkLogs

```
public List<NetworkEvent> retrieveNetworkLogs (ComponentName admin,
                long batchToken)
```

Called by device owner, profile owner of a managed profile or delegated app with `DELEGATION_NETWORK_LOGGING` to retrieve the most recent batch of network logging events.

When network logging is enabled by a profile owner, the network logs will only include work profile network activity, not activity on the personal profile. A device owner or profile owner has to provide a batchToken provided as part of `DeviceAdminReceiver.onNetworkLogsAvailable` callback. If the token doesn't match the token of the most recent available batch of logs, `null` will be returned.

`NetworkEvent` can be one of `DnsEvent` or `ConnectEvent`.

The list of network events is sorted chronologically, and contains at most 1200 events.

Access to the logs is rate limited and this method will only return a new batch of logs after the device device owner has been notified via `DeviceAdminReceiver.onNetworkLogsAvailable`.

If the caller is not a profile owner and a secondary user or profile is created, calling this method will throw a `SecurityException` until all users become affiliated again. It will also no longer be possible to retrieve the network logs batch with the most recent batchToken provided by `DeviceAdminReceiver.onNetworkLogsAvailable`. See `DevicePolicyManager.setAffiliationIds`.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with, or `null` if called by a delegated app.
`batchToken`	`long`: A token of the batch to retrieve
Returns	
`List<NetworkEvent>`	A new batch of network logs which is a list of `NetworkEvent`. Returns `null` if the batch represented by batchToken is no longer available or if logging is disabled.
Throws	
`SecurityException`	if `admin` is not a device owner, profile owner or if the `admin` is not a profile owner and there is at least one profile or secondary user that is not affiliated with the device.

retrievePreRebootSecurityLogs

```
public List<SecurityLog.SecurityEvent> retrievePreRebootSecurityLogs (ComponentName admin)
```

Called by device owner or profile owner of an organization-owned managed profile to retrieve device logs from before the device's last reboot.

This API is not supported on all devices. Calling this API on unsupported devices will result in `null` being returned. The device logs are retrieved from a RAM region which is not guaranteed to be corruption-free during power cycles, as a result be cautious about data corruption when parsing.

When called by a device owner, if there is any other user or profile on the device, it must be affiliated with the device. Otherwise a `SecurityException` will be thrown. See `isAffiliatedUser()`.

Parameters	
`admin`	`ComponentName`: Which device admin this request is associated with, or `null` if called by a delegated app.

Returns	
`List<SecurityLog.SecurityEvent>`	Device logs from before the latest reboot of the system, or `null` if this API is not supported on the device.

Throws	
`SecurityException`	if the caller is not allowed to access security logging, or there is at least one profile or secondary user that is not affiliated with the device.

retrieveSecurityLogs

```
public List<SecurityLog.SecurityEvent> retrieveSecurityLogs (ComponentName admin)
```

Called by device owner or profile owner of an organization-owned managed profile to retrieve all new security logging entries since the last call to this API after device boots.

Access to the logs is rate limited and it will only return new logs after the admin has been notified via `DeviceAdminReceiver.onSecurityLogsAvailable`.

When called by a device owner, if there is any other user or profile on the device, it must be affiliated with the device. Otherwise a `SecurityException` will be thrown. See `isAffiliatedUser()`.

Parameters	
`admin`	`ComponentName`: Which device admin this request is associated with, or `null` if called by a delegated app.

Returns	
`List<SecurityLog.SecurityEvent>`	the new batch of security logs which is a list of `SecurityEvent`, or `null` if rate limitation is exceeded or if logging is currently disabled.

Throws	

SecurityException	if the caller is not allowed to access security logging, or there is at least one profile or secondary user that is not affiliated with the device.

revokeKeyPairFromApp

```
public boolean revokeKeyPairFromApp (ComponentName admin,
                String alias,
                String packageName)
```

Called by a device or profile owner, or delegated certificate chooser (an app that has been delegated the DELEGATION_CERT_SELECTION privilege), to revoke an application's grant to a KeyChain key pair. Calls by the application to KeyChain.getPrivateKey(Context, String) will fail after the grant is revoked. The grantee app will receive the KeyChain.ACTION_KEY_ACCESS_CHANGED broadcast when access to a key is revoked. Starting from Build.VERSION_CODES.UPSIDE_DOWN_CAKE throws an IllegalArgumentException if alias doesn't correspond to an existing key.

Parameters	
admin	ComponentName : Which DeviceAdminReceiver this request is associated with, or null if calling from a delegated certificate chooser.
alias	String : The alias of the key to revoke access from. This value cannot be null .
packageName	String : The name of the (already installed) package to revoke access from. This value cannot be null .

Returns	
boolean	true if the grant was revoked successfully, false otherwise.

Throws	
IllegalArgumentException	if packageName or alias are empty, or if packageName is not a name of an installed package.
SecurityException	if the caller is not a device owner, a profile owner or delegated certificate chooser.

revokeKeyPairFromWifiAuth

```
public boolean revokeKeyPairFromWifiAuth (String alias)
```

Called by a device or profile owner, or delegated certificate chooser (an app that has been delegated the DELEGATION_CERT_SELECTION privilege), to deny using a KeyChain key pair for authentication to Wifi networks. Configured networks using this key won't be able to authenticate. Starting from Build.VERSION_CODES.UPSIDE_DOWN_CAKE throws an IllegalArgumentException if alias doesn't correspond to an existing key.

Parameters	
`alias`	`String` : The alias of the key pair. This value cannot be `null` .
Returns	
`boolean`	`true` if the operation was set successfully, `false` otherwise.
Throws	
`SecurityException`	if the caller is not a device owner, a profile owner or delegated certificate chooser.

setAccountManagementDisabled

```
public void setAccountManagementDisabled (ComponentName admin,
                String accountType,
                boolean disabled)
```

Called by a device owner or profile owner to disable account management for a specific type of account.

The calling device admin must be a device owner or profile owner. If it is not, a security exception will be thrown.

When account management is disabled for an account type, adding or removing an account of that type will not be possible.

From `Build.VERSION_CODES.N` the profile or device owner can still use `AccountManager` APIs to add or remove accounts when account management for a specific type is disabled.

This method may be called on the `DevicePolicyManager` instance returned from `getParentProfileInstance(ComponentName)` by the profile owner on an organization-owned device, to restrict accounts that may not be managed on the primary profile.

Starting from `Build.VERSION_CODES.UPSIDE_DOWN_CAKE` , after the account management disabled policy has been set, `PolicyUpdateReceiver.onPolicySetResult(Context,String,Bundle,TargetUser,PolicyUpdateResult)` will notify the admin on whether the policy was successfully set or not. This callback will contain:

- The policy identifier `DevicePolicyIdentifiers.ACCOUNT_MANAGEMENT_DISABLED_POLICY`
- The additional policy params bundle, which contains `PolicyUpdateReceiver.EXTRA_ACCOUNT_TYPE` the account type the policy applies to
- The `TargetUser` that this policy relates to
- The `PolicyUpdateResult` , which will be `PolicyUpdateResult.RESULT_POLICY_SET` if the policy was successfully set or the reason the policy failed to be set (e.g. `PolicyUpdateResult.RESULT_FAILURE_CONFLICTING_ADMIN_POLICY`)

If there has been a change to the policy, `PolicyUpdateReceiver.onPolicyChanged(Context,String,Bundle,TargetUser,PolicyUpdateResult)` will notify the

admin of this change. This callback will contain the same parameters as PolicyUpdateReceiver#onPolicySetResult and the `PolicyUpdateResult` will contain the reason why the policy changed.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null` .
`accountType`	`String` : For which account management is disabled or enabled.
`disabled`	`boolean` : The boolean indicating that account management will be disabled (true) or enabled (false).

Throws	
`SecurityException`	if `admin` is not a device or profile owner.

setAffiliationIds

```
public void setAffiliationIds (ComponentName admin,
                Set<String> ids)
```

Indicates the entity that controls the device. Two users are affiliated if the set of ids set by the device owner and the admin of the secondary user.

A user that is affiliated with the device owner user is considered to be affiliated with the device.

Note: Features that depend on user affiliation (such as security logging or `bindDeviceAdminServiceAsUser(ComponentName, Intent, ServiceConnection, BindServiceFlags, UserHandle)`) won't be available when a secondary user is created, until it becomes affiliated. Therefore it is recommended that the appropriate affiliation ids are set by its owner as soon as possible after the user is created.

Note: This method used to be available for affiliating device owner and profile owner. However, since Android 11, this combination is not possible. This method is now only useful for affiliating the primary user with managed secondary users.

Parameters	
`admin`	`ComponentName` : Which device owner, or owner of secondary user, this request is associated with. This value cannot be `null` .
`ids`	`Set` : A set of opaque non-empty affiliation ids. This value cannot be `null` .

Throws

`IllegalArgumentException`	if `ids` is null or contains an empty string.

setAlwaysOnVpnPackage

```
public void setAlwaysOnVpnPackage (ComponentName admin,
                String vpnPackage,
                boolean lockdownEnabled)
```

Called by a device or profile owner to configure an always-on VPN connection through a specific application for the current user. This connection is automatically granted and persisted after a reboot.

To support the always-on feature, an app must

- declare a `VpnService` in its manifest, guarded by `Manifest.permission.BIND_VPN_SERVICE`;
- target `API 24` or above; and
- *not* explicitly opt out of the feature through `VpnService.SERVICE_META_DATA_SUPPORTS_ALWAYS_ON`.

The call will fail if called with the package name of an unsupported VPN app.

Enabling lockdown via `lockdownEnabled` argument carries the risk that any failure of the VPN provider could break networking for all apps. This method clears any lockdown allowlist set by `setAlwaysOnVpnPackage(ComponentName,String,boolean,Set)`.

Starting from `API 31` calling this method with `vpnPackage` set to `null` only removes the existing configuration if it was previously created by this admin. To remove VPN configuration created by the user use `UserManager.DISALLOW_CONFIG_VPN`.

Parameters	
`admin`	`ComponentName`: This value cannot be `null`.
`vpnPackage`	`String`: The package name for an installed VPN app on the device, or `null` to remove an existing always-on VPN configuration.
`lockdownEnabled`	`boolean`: `true` to disallow networking when the VPN is not connected or `false` otherwise. This has no effect when clearing.

Throws	
`PackageManager.NameNotFoundException`	if `vpnPackage` is not installed.
`SecurityException`	if `admin` is not a device or a profile owner.
`UnsupportedOperationException`	if `vpnPackage` exists but does not support being set as always-on, or if always-on VPN is not available.

setAlwaysOnVpnPackage

```
public void setAlwaysOnVpnPackage (ComponentName admin,
                String vpnPackage,
                boolean lockdownEnabled,
                Set<String> lockdownAllowlist)
```

A version of `setAlwaysOnVpnPackage(ComponentName,String,boolean)` that allows the admin to specify a set of apps that should be able to access the network directly when VPN is not connected. When VPN connects these apps switch over to VPN if allowed to use that VPN. System apps can always bypass VPN.

Note that the system doesn't update the allowlist when packages are installed or uninstalled, the admin app must call this method to keep the list up to date.

When `lockdownEnabled` is false `lockdownAllowlist` is ignored . When `lockdownEnabled` is `true` and `lockdownAllowlist` is `null` or empty, only system apps can bypass VPN.

Setting always-on VPN package to `null` or using `setAlwaysOnVpnPackage(ComponentName,String,boolean)` clears lockdown allowlist.

Parameters	
`admin`	`ComponentName` : This value cannot be `null` .
`vpnPackage`	`String` : package name for an installed VPN app on the device, or `null` to remove an existing always-on VPN configuration
`lockdownEnabled`	`boolean` : `true` to disallow networking when the VPN is not connected or `false` otherwise. This has no effect when clearing.
`lockdownAllowlist`	`Set` : Packages that will be able to access the network directly when VPN is in lockdown mode but not connected. Has no effect when clearing. This value may be `null` .

Throws	
`PackageManager.NameNotFoundException`	if `vpnPackage` or one of `lockdownAllowlist` is not installed.
`SecurityException`	if `admin` is not a device or a profile owner.
`UnsupportedOperationException`	if `vpnPackage` exists but does not support being set as always-on, or if always-on VPN is not available.

setAppFunctionsPolicy

```
public void setAppFunctionsPolicy (int policy)
```

Sets the `AppFunctionManager` policy which controls app functions operations on the device. An app function is a piece of functionality that apps expose to the system for cross-app orchestration.

This function can only be called by a device owner, a profile owner or holders of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_APP_FUNCTIONS`.

Parameters	
`policy`	`int`: The app functions policy to set. One of `APP_FUNCTIONS_NOT_CONTROLLED_BY_POLICY`, `APP_FUNCTIONS_DISABLED` or `APP_FUNCTIONS_DISABLED_CROSS_PROFILE` Value is one of the following: • `APP_FUNCTIONS_NOT_CONTROLLED_BY_POLICY` • `APP_FUNCTIONS_DISABLED` • `APP_FUNCTIONS_DISABLED_CROSS_PROFILE`

Throws	
`SecurityException`	if caller is not a device owner, a profile owner or a holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_APP_FUNCTIONS`.

setApplicationHidden

```
public boolean setApplicationHidden (ComponentName admin,
                String packageName,
                boolean hidden)
```

Hide or unhide packages. When a package is hidden it is unavailable for use, but the data and actual package file remain. This function can be called by a device owner, profile owner, or by a delegate given the `DELEGATION_PACKAGE_ACCESS` scope via `setDelegatedScopes(ComponentName, String, List)`.

This method can be called on the `DevicePolicyManager` instance, returned by `getParentProfileInstance(ComponentName)`, where the caller must be the profile owner of an organization-owned managed profile and the package must be a system package. If called on the parent instance, then the package is hidden or unhidden in the personal profile.

Starting from `Build.VERSION_CODES.UPSIDE_DOWN_CAKE`, after the application hidden policy has been set, `PolicyUpdateReceiver.onPolicySetResult(Context,String,Bundle,TargetUser,PolicyUpdateResult)` will notify the admin on whether the policy was successfully set or not. This callback will contain:

- The policy identifier `DevicePolicyIdentifiers.APPLICATION_HIDDEN_POLICY`
- The additional policy params bundle, which contains `PolicyUpdateReceiver.EXTRA_PACKAGE_NAME` the package name the policy applies to
- The `TargetUser` that this policy relates to
- The `PolicyUpdateResult`, which will be `PolicyUpdateResult.RESULT_POLICY_SET` if the policy was successfully set or the reason the policy failed to be set (e.g. `PolicyUpdateResult.RESULT_FAILURE_CONFLICTING_ADMIN_POLICY`)

If there has been a change to the policy, `PolicyUpdateReceiver.onPolicyChanged(Context,String,Bundle,TargetUser,PolicyUpdateResult)` will notify the

admin of this change. This callback will contain the same parameters as PolicyUpdateReceiver#onPolicySetResult and the `PolicyUpdateResult` will contain the reason why the policy changed.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with, or `null` if the caller is not a device admin.
`packageName`	`String` : The name of the package to hide or unhide.
`hidden`	`boolean` : `true` if the package should be hidden, `false` if it should be unhidden.
Returns	
`boolean`	boolean Whether the hidden setting of the package was successfully updated.
Throws	
`IllegalArgumentException`	if called on the parent profile and the package provided is not a system package.
`SecurityException`	if `admin` is not a device or profile owner or if called on the parent profile and the `admin` is not a profile owner of an organization-owned managed profile.

setApplicationRestrictions

```
public void setApplicationRestrictions (ComponentName admin,
                String packageName,
                Bundle settings)
```

Sets the application restrictions for a given target application running in the calling user.

The caller must be a profile or device owner on that user, or the package allowed to manage application restrictions via `setDelegatedScopes(ComponentName, String, List)` with the `DELEGATION_APP_RESTRICTIONS` scope; otherwise a security exception will be thrown.

The provided `Bundle` consists of key-value pairs, where the types of values may be:

- `boolean`
- `int`
- `String` or `String[]`
- From `Build.VERSION_CODES.M` , `Bundle` or `Bundle[]`

If the restrictions are not available yet, but may be applied in the near future, the caller can notify the target application of that by adding `UserManager.KEY_RESTRICTIONS_PENDING` to the settings parameter.

The application restrictions are only made visible to the target application via `UserManager.getApplicationRestrictions(String)` , in addition to the profile or device owner, and the application

restrictions managing package via `getApplicationRestrictions(ComponentName, String)`.

Starting from Android Version `Build.VERSION_CODES.UPSIDE_DOWN_CAKE`, multiple admins can set app restrictions for the same application, the target application can get the list of app restrictions set by each admin via `RestrictionsManager.getApplicationRestrictionsPerAdmin()`.

Starting from Android Version `Build.VERSION_CODES.VANILLA_ICE_CREAM`, the device policy management role holder can also set app restrictions on any applications in the calling user, as well as the parent user of an organization-owned managed profile via the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)`. App restrictions set by the device policy management role holder are returned by `UserManager.getApplicationRestrictions(String)` but the target application should use `RestrictionsManager.getApplicationRestrictionsPerAdmin()` to retrieve them, alongside any app restrictions the profile or device owner might have set.

NOTE: The method performs disk I/O and shouldn't be called on the main thread.
This method may take several seconds to complete, so it should only be called from a worker thread.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with, or `null` if called by the application restrictions managing package.
`packageName`	`String`: The name of the package to update restricted settings for.
`settings`	`Bundle`: A `Bundle` to be parsed by the receiving application, conveying a new set of active restrictions.

Throws	
`SecurityException`	if `admin` is not a device or profile owner.

See also:

- `setDelegatedScopes(ComponentName, String, List)`
- `DELEGATION_APP_RESTRICTIONS`
- `UserManager.KEY_RESTRICTIONS_PENDING`

setApplicationRestrictionsManagingPackage

```
public void setApplicationRestrictionsManagingPackage (ComponentName admin,
                String packageName)
```

This method was deprecated in API level 26.
From `Build.VERSION_CODES.O`. Use `setDelegatedScopes(ComponentName, String, List)` with the `DELEGATION_APP_RESTRICTIONS` scope instead.

Called by a profile owner or device owner to grant permission to a package to manage application restrictions for the calling user via `setApplicationRestrictions(ComponentName, String, Bundle)` and `getApplicationRestrictions(ComponentName, String)`.

This permission is persistent until it is later cleared by calling this method with a `null` value or uninstalling the managing package.

The supplied application restriction managing package must be installed when calling this API, otherwise an `NameNotFoundException` will be thrown.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null`.
`packageName`	`String`: The package name which will be given access to application restrictions APIs. If `null` is given the current package will be cleared.

Throws	
`PackageManager.NameNotFoundException`	if `packageName` is not found
`SecurityException`	if `admin` is not a device or profile owner.

setAutoTimeEnabled

```
public void setAutoTimeEnabled (ComponentName admin,
                boolean enabled)
```

Called by a device owner, a profile owner for the primary user or a profile owner of an organization-owned managed profile to turn auto time on and off.

Callers are recommended to use `UserManager.DISALLOW_CONFIG_DATE_TIME` to prevent the user from changing this setting, that way no user will be able set the date and time zone.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null`.
`enabled`	`boolean`: Whether time should be obtained automatically from the network or not.

Throws	
`SecurityException`	if caller is not a device owner, a profile owner for the primary user, or a profile owner of an organization-owned managed profile.

setAutoTimePolicy

```
public void setAutoTimePolicy (int policy)
```

Called by a device owner, a profile owner for the primary user or a profile owner of an organization-owned managed profile to turn auto time on and off i.e. Whether time should be obtained automatically from the network or not.

Callers are recommended to use `UserManager.DISALLOW_CONFIG_DATE_TIME` to prevent the user from changing this setting, that way no user will be able set the date and time zone.

Parameters	
`policy`	`int` : The desired state among `AUTO_TIME_ENABLED` to enable, `AUTO_TIME_DISABLED` to disable and `AUTO_TIME_NOT_CONTROLLED_BY_POLICY` to unset the policy. Value is one of the following: • `AUTO_TIME_NOT_CONTROLLED_BY_POLICY` • `AUTO_TIME_DISABLED` • `AUTO_TIME_ENABLED`

Throws	
`SecurityException`	if caller is not a device owner, a profile owner for the primary user, or a profile owner of an organization-owned managed profile, or if the caller does not hold the required permission.

setAutoTimeRequired

```
public void setAutoTimeRequired (ComponentName admin,
                boolean required)
```

This method was deprecated in API level 30.
From `Build.VERSION_CODES.R` . Use `setAutoTimeEnabled(ComponentName, boolean)` to turn auto time on or off and use `UserManager.DISALLOW_CONFIG_DATE_TIME` to prevent the user from changing this setting.

Called by a device owner, or alternatively a profile owner from Android 8.0 (API level 26) or higher, to set whether auto time is required. If auto time is required, no user will be able set the date and time and network date and time will be used.

Note: If auto time is required the user can still manually set the time zone. Staring from Android 11, if auto time is required, the user cannot manually set the time zone.

The calling device admin must be a device owner, or alternatively a profile owner from Android 8.0 (API level 26) or higher. If it is not, a security exception will be thrown.

Staring from Android 11, this API switches to use `UserManager.DISALLOW_CONFIG_DATE_TIME` to enforce the auto time settings. Calling this API to enforce auto time will result in `UserManager.DISALLOW_CONFIG_DATE_TIME` being set, while calling this API to lift the requirement will result in `UserManager.DISALLOW_CONFIG_DATE_TIME` being cleared. From Android 11, this API can also no longer be called on a managed profile.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
`required`	`boolean` : Whether auto time is set required or not.
Throws	
`SecurityException`	if `admin` is not a device owner, not a profile owner or if this API is called on a managed profile.

setAutoTimeZoneEnabled

```
public void setAutoTimeZoneEnabled (ComponentName admin,
                boolean enabled)
```

Called by a device owner, a profile owner for the primary user or a profile owner of an organization-owned managed profile to turn auto time zone on and off.

Callers are recommended to use `UserManager.DISALLOW_CONFIG_DATE_TIME` to prevent the user from changing this setting, that way no user will be able set the date and time zone.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with or Null if the caller is not a device admin. This value may be `null` .
`enabled`	`boolean` : Whether time zone should be obtained automatically from the network or not.
Throws	
`SecurityException`	if caller is not a device owner, a profile owner for the primary user, or a profile owner of an organization-owned managed profile.

setAutoTimeZonePolicy

```
public void setAutoTimeZonePolicy (int policy)
```

Called by a device owner, a profile owner for the primary user or a profile owner of an organization-owned managed profile to turn auto time zone on and off.

Callers are recommended to use `UserManager.DISALLOW_CONFIG_DATE_TIME` to prevent the user from changing this setting, that way no user will be able set the date and time zone.

Parameters

`policy`	`int` : The desired state among `AUTO_TIME_ZONE_ENABLED` to enable it, `AUTO_TIME_ZONE_DISABLED` to disable it or `AUTO_TIME_ZONE_NOT_CONTROLLED_BY_POLICY` to unset the policy. Value is one of the following: • `AUTO_TIME_ZONE_NOT_CONTROLLED_BY_POLICY` • `AUTO_TIME_ZONE_DISABLED` • `AUTO_TIME_ZONE_ENABLED`

Throws	
`SecurityException`	if caller is not a device owner, a profile owner for the primary user, or a profile owner of an organization-owned managed profile, or if the caller does not hold the required permission.

setBackupServiceEnabled

```
public void setBackupServiceEnabled (ComponentName admin,
                boolean enabled)
```

Allows the device owner or profile owner to enable or disable the backup service.

Each user has its own backup service which manages the backup and restore mechanisms in that user. Disabling the backup service will prevent data from being backed up or restored.

Device owner calls this API to control backup services across all users on the device. Profile owner can use this API to enable or disable the profile's backup service. However, for a managed profile its backup functionality is only enabled if both the device owner and the profile owner have enabled the backup service.

By default, backup service is disabled on a device with device owner, and within a managed profile.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
`enabled`	`boolean` : `true` to enable the backup service, `false` to disable it.

Throws	
`SecurityException`	if `admin` is not a device owner or a profile owner.

setBluetoothContactSharingDisabled

```
public void setBluetoothContactSharingDisabled (ComponentName admin,
                boolean disabled)
```

Called by a profile owner of a managed profile to set whether bluetooth devices can access enterprise contacts.

The calling device admin must be a profile owner. If it is not, a security exception will be thrown.

This API works on managed profile only.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
`disabled`	`boolean` : If true, bluetooth devices cannot access enterprise contacts.
Throws	
`SecurityException`	if `admin` is not a profile owner.

setCameraDisabled

```
public void setCameraDisabled (ComponentName admin,
                boolean disabled)
```

Called by an application that is administering the device to disable all cameras on the device, for this user. After setting this, no applications running as this user will be able to access any cameras on the device.

Starting with Android `Build.VERSION_CODES.CINNAMON_BUN` , this method also blocks application access to external USB cameras that connect directly via the `UsbConstants.USB_CLASS_VIDEO` interface.

This method can be called on the `DevicePolicyManager` instance, returned by `getParentProfileInstance(ComponentName)` , where the caller must be the profile owner of an organization-owned managed profile.

If the caller is device owner, then the restriction will be applied to all users. If called on the parent instance, then the restriction will be applied on the personal profile.

The calling device admin must have requested `DeviceAdminInfo.USES_POLICY_DISABLE_CAMERA` to be able to call this method; if it has not, a security exception will be thrown.

Note, this policy type is deprecated for legacy device admins since `Build.VERSION_CODES.Q` . On Android `Build.VERSION_CODES.Q` devices, legacy device admins targeting SDK version `Build.VERSION_CODES.P` or below can still call this API to disable camera, while legacy device admins targeting SDK version `Build.VERSION_CODES.Q` will receive a SecurityException. Starting from Android `Build.VERSION_CODES.R` , requests to disable camera from legacy device admins targeting SDK version `Build.VERSION_CODES.P` or below will be silently ignored.

Starting from `Build.VERSION_CODES.UPSIDE_DOWN_CAKE` , after the camera disabled policy has been set, `PolicyUpdateReceiver.onPolicySetResult(Context,String,Bundle,TargetUser,PolicyUpdateResult)` will notify the admin on whether the policy was successfully set or not. This callback will contain:

- The policy identifier: userRestriction_no_camera
- The `TargetUser` that this policy relates to

- The `PolicyUpdateResult`, which will be `PolicyUpdateResult.RESULT_POLICY_SET` if the policy was successfully set or the reason the policy failed to be set (e.g. `PolicyUpdateResult.RESULT_FAILURE_CONFLICTING_ADMIN_POLICY`)

If there has been a change to the policy, `PolicyUpdateReceiver.onPolicyChanged(Context,String,Bundle,TargetUser,PolicyUpdateResult)` will notify the admin of this change. This callback will contain the same parameters as PolicyUpdateReceiver#onPolicySetResult and the `PolicyUpdateResult` will contain the reason why the policy changed.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with or null if the caller is not a device admin
`disabled`	`boolean`: Whether or not the camera should be disabled.
Throws	
`SecurityException`	if `admin` is not an active administrator or does not use `DeviceAdminInfo.USES_POLICY_DISABLE_CAMERA`.

setCertInstallerPackage

```
public void setCertInstallerPackage (ComponentName admin,
                String installerPackage)
```

This method was deprecated in API level 26.
From `Build.VERSION_CODES.O`. Use `setDelegatedScopes(ComponentName, String, List)` with the `DELEGATION_CERT_INSTALL` scope instead.

Called by a profile owner or device owner to grant access to privileged certificate manipulation APIs to a third-party certificate installer app. Granted APIs include `getInstalledCaCerts(ComponentName)`, `hasCaCertInstalled(ComponentName, byte)`, `installCaCert(ComponentName, byte)`, `uninstallCaCert(ComponentName, byte)`, `uninstallAllUserCaCerts(ComponentName)` and `installKeyPair(ComponentName, PrivateKey, Certificate, String)`.

Delegated certificate installer is a per-user state. The delegated access is persistent until it is later cleared by calling this method with a null value or uninstalling the certificate installer.

Note: Starting from `Build.VERSION_CODES.N`, if the caller application's target SDK version is `Build.VERSION_CODES.N` or newer, the supplied certificate installer package must be installed when calling this API, otherwise an `IllegalArgumentException` will be thrown.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null`.

`installerPackage`	`String` : The package name of the certificate installer which will be given access. If `null` is given the current package will be cleared.

Throws	
`SecurityException`	if `admin` is not a device or a profile owner.

setCommonCriteriaModeEnabled

```
public void setCommonCriteriaModeEnabled (ComponentName admin,
                boolean enabled)
```

Called by device owner or profile owner of an organization-owned managed profile to toggle Common Criteria mode for the device. When the device is in Common Criteria mode, certain device functionalities are tuned to meet the higher security level required by Common Criteria certification. For example:

- Bluetooth long term key material is additionally integrity-protected with AES-GCM.
- WiFi configuration store is additionally integrity-protected with AES-GCM.

Common Criteria mode is disabled by default.

Note: if Common Critera mode is turned off after being enabled previously, all existing WiFi configurations will be lost.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null` .
`enabled`	`boolean` : whether Common Criteria mode should be enabled or not.

setConfiguredNetworksLockdownState

```
public void setConfiguredNetworksLockdownState (ComponentName admin,
                boolean lockdown)
```

Called by a device owner or a profile owner of an organization-owned managed profile to control whether the user can change networks configured by the admin. When this lockdown is enabled, the user can still configure and connect to other Wi-Fi networks, or use other Wi-Fi capabilities such as tethering.

WiFi network configuration lockdown is controlled by a global settings `Settings.Global.WIFI_DEVICE_OWNER_CONFIGS_LOCKDOWN` and calling this API effectively modifies the global settings. Previously device owners can also control this directly via `setGlobalSetting(ComponentName, String, String)` but they are recommended to switch to this API.

Parameters

`admin`	`ComponentName` : admin Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null` .
`lockdown`	`boolean` : Whether the admin configured networks should be unmodifiable by the user.

Throws	
`SecurityException`	if caller is not a device owner or a profile owner of an organization-owned managed profile.

setContentProtectionPolicy

```
public void setContentProtectionPolicy (ComponentName admin,
                int policy)
```

Sets the content protection policy which controls scanning for deceptive apps.

This function can only be called by the device owner, a profile owner of an affiliated user or profile, or the profile owner when no device owner is set or holders of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_CONTENT_PROTECTION` . See `isAffiliatedUser()` . Any policy set via this method will be cleared if the user becomes unaffiliated.

After the content protection policy has been set, `PolicyUpdateReceiver.onPolicySetResult(Context,String,Bundle,TargetUser,PolicyUpdateResult)` will notify the admin on whether the policy was successfully set or not. This callback will contain:

- The policy identifier `DevicePolicyIdentifiers.CONTENT_PROTECTION_POLICY`
- The `TargetUser` that this policy relates to
- The `PolicyUpdateResult` , which will be `PolicyUpdateResult.RESULT_POLICY_SET` if the policy was successfully set or the reason the policy failed to be set (e.g. `PolicyUpdateResult.RESULT_FAILURE_CONFLICTING_ADMIN_POLICY`)

If there has been a change to the policy, `PolicyUpdateReceiver.onPolicyChanged(Context,String,Bundle,TargetUser,PolicyUpdateResult)` will notify the admin of this change. This callback will contain the same parameters as PolicyUpdateReceiver#onPolicySetResult and the `PolicyUpdateResult` will contain the reason why the policy changed.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null` .
`policy`	`int` : The content protection policy to set. One of `CONTENT_PROTECTION_NOT_CONTROLLED_BY_POLICY` , `CONTENT_PROTECTION_DISABLED` or `CONTENT_PROTECTION_ENABLED` . Value is one of the following:

	• CONTENT_PROTECTION_NOT_CONTROLLED_BY_POLICY • CONTENT_PROTECTION_DISABLED • CONTENT_PROTECTION_ENABLED

Throws	
SecurityException	if admin is not the device owner, the profile owner of an affiliated user or profile, or the profile owner when no device owner is set or holder of the permission Manifest.permission.MANAGE_DEVICE_POLICY_CONTENT_PROTECTION .

setCredentialManagerPolicy

```
public void setCredentialManagerPolicy (PackagePolicy policy)
```

Called by a device owner or profile owner of a managed profile to set the credential manager policy.

Affects APIs exposed by CredentialManager .

A PackagePolicy.PACKAGE_POLICY_ALLOWLIST policy type will limit the credential providers that the user can use to the list of packages in the policy.

A PackagePolicy.PACKAGE_POLICY_ALLOWLIST_AND_SYSTEM policy type allows access from the OEM default credential providers and the allowlist of credential providers.

A PackagePolicy.PACKAGE_POLICY_BLOCKLIST policy type will block the credential providers listed in the policy from being used by the user.

Parameters	
policy	PackagePolicy : the policy to set, setting this value to null will allow all packages

Throws	
SecurityException	if caller is not a device owner or profile owner of a managed profile

setCrossProfileCalendarPackages

```
public void setCrossProfileCalendarPackages (ComponentName admin,
                Set<String> packageNames)
```

This method was deprecated in API level 34.
Use setCrossProfilePackages(ComponentName,Set) .

Allows a set of packages to access cross-profile calendar APIs.

Called by a profile owner of a managed profile.

Calling with a `null` value for the set disables the restriction so that all packages are allowed to access cross-profile calendar APIs. Calling with an empty set disallows all packages from accessing cross-profile calendar APIs. If this method isn't called, no package is allowed to access cross-profile calendar APIs by default.

Parameters	
`admin`	`ComponentName` : which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
`packageNames`	`Set` : set of packages to be allowlisted. This value may be `null` .
Throws	
`SecurityException`	if `admin` is not a profile owner

setCrossProfileCallerIdDisabled

```
public void setCrossProfileCallerIdDisabled (ComponentName admin,
                boolean disabled)
```

This method was deprecated in API level 34.
starting with `Build.VERSION_CODES.UPSIDE_DOWN_CAKE` , use `setManagedProfileCallerIdAccessPolicy(PackagePolicy)` instead

Called by a profile owner of a managed profile to set whether caller-Id information from the managed profile will be shown in the parent profile, for incoming calls.

The calling device admin must be a profile owner. If it is not, a security exception will be thrown.

Starting with `Build.VERSION_CODES.UPSIDE_DOWN_CAKE` , calling this function is similar to calling `setManagedProfileCallerIdAccessPolicy(PackagePolicy)` with a `PackagePolicy.PACKAGE_POLICY_BLOCKLIST` policy type when `disabled` is false or a `PackagePolicy.PACKAGE_POLICY_ALLOWLIST` policy type when `disabled` is true.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
`disabled`	`boolean` : If true caller-Id information in the managed profile is not displayed.
Throws	
`SecurityException`	if `admin` is not a profile owner.

setCrossProfileContactsSearchDisabled

```
public void setCrossProfileContactsSearchDisabled (ComponentName admin,
                boolean disabled)
```

This method was deprecated in API level 34.
From `Build.VERSION_CODES.UPSIDE_DOWN_CAKE` use `setManagedProfileContactsAccessPolicy(PackagePolicy)`

Called by a profile owner of a managed profile to set whether contacts search from the managed profile will be shown in the parent profile, for incoming calls.

The calling device admin must be a profile owner. If it is not, a security exception will be thrown. Starting with `Build.VERSION_CODES.UPSIDE_DOWN_CAKE`, calling this function is similar to calling `setManagedProfileContactsAccessPolicy(PackagePolicy)` with a `PackagePolicy.PACKAGE_POLICY_BLOCKLIST` policy type when `disabled` is false or a `PackagePolicy.PACKAGE_POLICY_ALLOWLIST` policy type when `disabled` is true.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null`.
`disabled`	`boolean`: If true contacts search in the managed profile is not displayed.
Throws	
`SecurityException`	if `admin` is not a profile owner.

setCrossProfilePackages

```
public void setCrossProfilePackages (ComponentName admin,
                Set<String> packageNames)
```

Sets the set of admin-allowlisted package names that are allowed to request user consent for cross-profile communication.

Assumes that the caller is a profile owner and is the given `admin`.

Previous calls are overridden by each subsequent call to this method.

Note that other apps may be able to request user consent for cross-profile communication if they have been explicitly allowlisted by the OEM.

When previously-set cross-profile packages are missing from `packageNames`, the app-op for `INTERACT_ACROSS_PROFILES` will be reset for those packages. This will not occur for packages that are allowlisted by the OEM.

Parameters	
`admin`	`ComponentName`: the `DeviceAdminReceiver` this request is associated with. This value cannot be `null`.

`packageNames`	`Set` : the new cross-profile package names. This value cannot be `null` .

setCrossProfileWidgetProviders

```
public void setCrossProfileWidgetProviders (Set<String> packageNames)
```

Called by the profile owner of a managed profile or a holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_PROFILE_INTERACTION` to enable widget providers from packages to be available in the parent profile. As a result the user will be able to add widgets from the allowlisted package running under the profile to a widget host which runs under the parent profile, for example the home screen. Note that a package may have zero or more provider components, where each component provides a different widget type.

Note: By default no widget provider package is allowlisted.

Note: This API updates the entire allowlist in one-go, overriding any previous allowlist. This is more efficient than using `addCrossProfileWidgetProvider(ComponentName, String)` and `removeCrossProfileWidgetProvider(ComponentName, String)` to update the allowlist one package a time, especially if the allowlist consists of many packages.

Parameters	
`packageNames`	`Set` : The packages from which widget providers are allowlisted. This value cannot be `null` .
Throws	
`SecurityException`	if `admin` is not a profile owner and not a holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_PROFILE_INTERACTION` .

setDefaultDialerApplication

```
public void setDefaultDialerApplication (String packageName)
```

Must be called by a device owner or a profile owner of an organization-owned managed profile to set the default dialer application for the calling user.

When the profile owner of an organization-owned managed profile calls this method, it sets the default dialer application in the work profile. This is only meaningful when work profile telephony is enabled by `setManagedSubscriptionsPolicy(ManagedSubscriptionsPolicy)` .

If the device does not support telephony (`PackageManager.FEATURE_TELEPHONY`), calling this method will do nothing.

Parameters

`packageName`	`String` : The name of the package to set as the default dialer application. This value cannot be `null` .
Throws	
`IllegalArgumentException`	if the package cannot be set as the default dialer, for example if the package is not installed or does not expose the expected activities or services that a dialer app is required to have.
`SecurityException`	if `admin` is not a device or profile owner or a profile owner of an organization-owned managed profile.

setDefaultSmsApplication

```
public void setDefaultSmsApplication (ComponentName admin,
                String packageName)
```

Must be called by a device owner or a profile owner of an organization-owned managed profile to set the default SMS application.

This method can be called on the `DevicePolicyManager` instance, returned by `getParentProfileInstance(ComponentName)` , where the caller must be the profile owner of an organization-owned managed profile and the package must be a pre-installed system package. If called on the parent instance, then the default SMS application is set on the personal profile.

Starting from Android `Build.VERSION_CODES.UPSIDE_DOWN_CAKE` , the profile owner of an organization-owned managed profile can also call this method directly (not on the parent profile instance) to set the default SMS application in the work profile. This is only meaningful when work profile telephony is enabled by `setManagedSubscriptionsPolicy(ManagedSubscriptionsPolicy)` .

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null` .
`packageName`	`String` : The name of the package to set as the default SMS application. This value cannot be `null` .
Throws	
`IllegalArgumentException`	if called on the parent profile and the package provided is not a pre-installed system package.
`IllegalStateException`	while trying to set default sms app on the profile and `ManagedSubscriptionsPolicy.TYPE_ALL_MANAGED_SUBSCRIPTIONS` policy is not set.
`SecurityException`	if `admin` is not a device or profile owner or if called on the parent profile and the `admin` is not a profile owner of an organization-owned managed profile.

setDelegatedScopes

```
public void setDelegatedScopes (ComponentName admin,
                String delegatePackage,
                List<String> scopes)
```

Called by a profile owner or device owner to grant access to privileged APIs to another app. Granted APIs are determined by `scopes`, which is a list of the `DELEGATION_*` constants.

A broadcast with the `ACTION_APPLICATION_DELEGATION_SCOPES_CHANGED` action will be sent to the `delegatePackage` with its new scopes in an `ArrayList<String>` extra under the `EXTRA_DELEGATION_SCOPES` key. The broadcast is sent with the `Intent.FLAG_RECEIVER_REGISTERED_ONLY` flag.

Delegated scopes are a per-user state. The delegated access is persistent until it is later cleared by calling this method with an empty `scopes` list or uninstalling the `delegatePackage`.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null`.
`delegatePackage`	`String`: The package name of the app which will be given access. This value cannot be `null`.
`scopes`	`List`: The groups of privileged APIs whose access should be granted to `delegatedPackage`. This value cannot be `null`.
Throws	
`SecurityException`	if `admin` is not a device or a profile owner.

setDeviceOwnerLockScreenInfo

```
public void setDeviceOwnerLockScreenInfo (ComponentName admin,
                CharSequence info)
```

Sets the device owner information to be shown on the lock screen.

Device owner information set using this method overrides any owner information manually set by the user and prevents the user from further changing it.

If the device owner information is `null` or empty then the device owner info is cleared and the user owner info is shown on the lock screen if it is set.

If the device owner information contains only whitespaces then the message on the lock screen will be blank and the user will not be allowed to change it.

If the device owner information needs to be localized, it is the responsibility of the `DeviceAdminReceiver` to listen to the `Intent.ACTION_LOCALE_CHANGED` broadcast and set a new version of this string accordingly.

May be called by the device owner or the profile owner of an organization-owned device.

Parameters	
`admin`	`ComponentName` : The name of the admin component to check. Null if the caller is not a device admin. This value may be `null` .
`info`	`CharSequence` : Device owner information which will be displayed instead of the user owner info.
Throws	
`SecurityException`	if `admin` is not a device owner.

setEndUserSessionMessage

```
public void setEndUserSessionMessage (ComponentName admin,
                CharSequence endUserSessionMessage)
```

Called by a device owner to specify the user session end message. This may be displayed during a user switch.

The message should be limited to a short statement or it may be truncated.

If the message needs to be localized, it is the responsibility of the `DeviceAdminReceiver` to listen to the `Intent.ACTION_LOCALE_CHANGED` broadcast and set a new version of this message accordingly.

Parameters	
`admin`	`ComponentName` : which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
`endUserSessionMessage`	`CharSequence` : message for ending user session, or `null` to use system default message.
Throws	
`SecurityException`	if `admin` is not a device owner.

setFactoryResetProtectionPolicy

```
public void setFactoryResetProtectionPolicy (ComponentName admin,
                FactoryResetProtectionPolicy policy)
```

Callable by device owner or profile owner of an organization-owned device, to set a factory reset protection (FRP) policy. When a new policy is set, the system notifies the FRP management agent of a policy change by broadcasting `ACTION_RESET_PROTECTION_POLICY_CHANGED` .

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null` .
`policy`	`FactoryResetProtectionPolicy` : the new FRP policy, or `null` to clear the current policy.

Throws	
`SecurityException`	if `admin` is not a device owner or a profile owner of an organization-owned device.
`UnsupportedOperationException`	if factory reset protection is not supported on the device.

setGlobalPrivateDnsModeOpportunistic

```
public int setGlobalPrivateDnsModeOpportunistic (ComponentName admin)
```

Sets the global Private DNS mode to opportunistic. May only be called by the device owner.

In this mode, the DNS subsystem will attempt a TLS handshake to the network-supplied resolver prior to attempting name resolution in cleartext.

Note: The device owner won't be able to set the global private DNS mode if there are unaffiliated secondary users or profiles on the device. It's recommended that affiliation ids are set for new users as soon as possible after provisioning via `setAffiliationIds(ComponentName, Set)` .

Parameters	
`admin`	`ComponentName` : which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .

Returns	
`int`	`PRIVATE_DNS_SET_NO_ERROR` if the mode was set successfully, or `PRIVATE_DNS_SET_ERROR_FAILURE_SETTING` if it could not be set. Value is one of the following: • `PRIVATE_DNS_SET_NO_ERROR` • `PRIVATE_DNS_SET_ERROR_HOST_NOT_SERVING` • `PRIVATE_DNS_SET_ERROR_FAILURE_SETTING`

Throws	
`SecurityException`	if the caller is not the device owner.

setGlobalPrivateDnsModeSpecifiedHost

```
public int setGlobalPrivateDnsModeSpecifiedHost (ComponentName admin,
                String privateDnsHost)
```

Sets the global Private DNS host to be used. May only be called by the device owner.

Note that the method is blocking as it will perform a connectivity check to the resolver, to ensure it is valid. Because of that, the method should not be called on any thread that relates to user interaction, such as the UI thread.

In case a VPN is used in conjunction with Private DNS resolver, the Private DNS resolver must be reachable both from within and outside the VPN. Otherwise, the device may lose the ability to resolve hostnames as system traffic to the resolver may not go through the VPN.

Note: The device owner won't be able to set the global private DNS mode if there are unaffiliated secondary users or profiles on the device. It's recommended that affiliation ids are set for new users as soon as possible after provisioning via `setAffiliationIds(ComponentName, Set)`.
This method may take several seconds to complete, so it should only be called from a worker thread.

Parameters	
`admin`	`ComponentName`: which `DeviceAdminReceiver` this request is associated with. This value cannot be `null`.
`privateDnsHost`	`String`: The hostname of a server that implements DNS over TLS (RFC7858). This value cannot be `null`.

Returns	
`int`	`PRIVATE_DNS_SET_NO_ERROR` if the mode was set successfully, `PRIVATE_DNS_SET_ERROR_FAILURE_SETTING` if it could not be set or `PRIVATE_DNS_SET_ERROR_HOST_NOT_SERVING` if the specified host does not implement RFC7858. Value is one of the following: • `PRIVATE_DNS_SET_NO_ERROR` • `PRIVATE_DNS_SET_ERROR_HOST_NOT_SERVING` • `PRIVATE_DNS_SET_ERROR_FAILURE_SETTING`

Throws	
`IllegalArgumentException`	if the `privateDnsHost` is not a valid hostname.
`SecurityException`	if the caller is not the device owner.

setGlobalSetting

```
public void setGlobalSetting (ComponentName admin,
                String setting,
```

```
                String value)
```

This method is mostly deprecated. Most of the settings that still have an effect have dedicated setter methods or user restrictions. See individual settings for details.

Called by device owner to update `Settings.Global` settings. Validation that the value of the setting is in the correct form for the setting type should be performed by the caller.

The settings that can be updated with this method are:

- `Settings.Global.ADB_ENABLED` : use `UserManager.DISALLOW_DEBUGGING_FEATURES` instead to restrict users from enabling debugging features and this setting to turn adb on.
- `Settings.Global.USB_MASS_STORAGE_ENABLED`
- `Settings.Global.STAY_ON_WHILE_PLUGGED_IN` This setting is only available from `Build.VERSION_CODES.M` onwards and can only be set if `setMaximumTimeToLock(ComponentName, long)` is not used to set a timeout.
- `Settings.Global.WIFI_DEVICE_OWNER_CONFIGS_LOCKDOWN`
  This setting is only available from `Build.VERSION_CODES.M` onwards.

The following settings used to be supported, but can be controlled in other ways:

- `Settings.Global.AUTO_TIME` : Use `setAutoTimeEnabled(ComponentName, boolean)` and `UserManager.DISALLOW_CONFIG_DATE_TIME` instead.
- `Settings.Global.AUTO_TIME_ZONE` : Use `setAutoTimeZoneEnabled(ComponentName, boolean)` and `UserManager.DISALLOW_CONFIG_DATE_TIME` instead.
- `Settings.Global.DATA_ROAMING` : Use `UserManager.DISALLOW_DATA_ROAMING` instead.

Changing the following settings has no effect as of `Build.VERSION_CODES.M` :

- `Settings.Global.BLUETOOTH_ON` . Use `BluetoothAdapter.enable()` and `BluetoothAdapter.disable()` instead.
- `Settings.Global.DEVELOPMENT_SETTINGS_ENABLED`
- `Settings.Global.MODE_RINGER` . Use `AudioManager.setRingerMode(int)` instead.
- `Settings.Global.NETWORK_PREFERENCE`
- `Settings.Global.WIFI_ON` . Use `WifiManager.setWifiEnabled(boolean)` instead.
- `Settings.Global.WIFI_SLEEP_POLICY` . No longer has effect.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
`setting`	`String` : The name of the setting to update.
`value`	`String` : The value to update the setting to.

Throws	
`SecurityException`	if `admin` is not a device owner.

setKeepUninstalledPackages

```
public void setKeepUninstalledPackages (ComponentName admin,
                List<String> packageNames)
```

Set a list of apps to keep around as APKs even if no user has currently installed it. This function can be called by a device owner or by a delegate given the `DELEGATION_KEEP_UNINSTALLED_PACKAGES` scope via `setDelegatedScopes(ComponentName, String, List)`.

Please note that setting this policy does not imply that specified apps will be automatically pre-cached.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with, or `null` if the caller is a keep uninstalled packages delegate.
`packageNames`	`List`: List of package names to keep cached. This value cannot be `null`.
Throws	
`SecurityException`	if `admin` is not a device owner.

setKeyPairCertificate

```
public boolean setKeyPairCertificate (ComponentName admin,
                String alias,
                List<Certificate> certs,
                boolean isUserSelectable)
```

This API can be called by the following to associate certificates with a key pair that was generated using `generateKeyPair(ComponentName, String, KeyGenParameterSpec, int)`, and set whether the key is available for the user to choose in the certificate selection prompt:

- Device owner
- Profile owner
- Delegated certificate installer
- Credential management app

From Android `Build.VERSION_CODES.S`, the credential management app can call this API. If called by the credential management app, the componentName must be `null`. Note, there can only be a credential management app on an unmanaged device.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with, or `null` if the caller is not a device admin.

`alias`	`String` : The private key alias under which to install the certificate. The `alias` should denote an existing private key. If a certificate with that alias already exists, it will be overwritten. This value cannot be `null` .
`certs`	`List` : The certificate chain to install. The chain should start with the leaf certificate and include the chain of trust in order. This will be returned by `KeyChain.getCertificateChain(Context, String)` . This value cannot be `null` .
`isUserSelectable`	`boolean` : `true` to indicate that a user can select this key via the certificate selection prompt, `false` to indicate that this key can only be granted access by implementing `DeviceAdminReceiver.onChoosePrivateKeyAlias(Context, Intent, int, Uri, String)` .
Returns	
`boolean`	`true` if the provided `alias` exists and the certificates has been successfully associated with it, `false` otherwise.
Throws	
`SecurityException`	if `admin` is not `null` and not a device or profile owner, or `admin` is null but the calling application is not a delegated certificate installer or credential management app.

setKeyguardDisabled

```
public boolean setKeyguardDisabled (ComponentName admin,
                boolean disabled)
```

Called by a device owner or profile owner of secondary users that is affiliated with the device to disable the keyguard altogether.

Setting the keyguard to disabled has the same effect as choosing "None" as the screen lock type. However, this call has no effect if a password, pin or pattern is currently set. If a password, pin or pattern is set after the keyguard was disabled, the keyguard stops being disabled.

As of `Build.VERSION_CODES.P` , this call also dismisses the keyguard if it is currently shown.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
`disabled`	`boolean` : `true` disables the keyguard, `false` reenables it.
Returns	
`boolean`	`false` if attempting to disable the keyguard while a lock password was in place. `true` otherwise.
Throws	

`SecurityException`	if `admin` is not the device owner, or a profile owner of secondary user that is affiliated with the device.

setKeyguardDisabledFeatures

```
public void setKeyguardDisabledFeatures (ComponentName admin,
                int which)
```

Called by an application that is administering the device to disable keyguard customizations, such as widgets. After setting this, keyguard features will be disabled according to the provided feature list.

A calling device admin must have requested `DeviceAdminInfo.USES_POLICY_DISABLE_KEYGUARD_FEATURES` to be able to call this method; if it has not, a security exception will be thrown.

Calling this from a managed profile before version `Build.VERSION_CODES.M` will throw a security exception. From version `Build.VERSION_CODES.M` the profile owner of a managed profile can set:

- `KEYGUARD_DISABLE_TRUST_AGENTS`, which affects the parent user, but only if there is no separate challenge set on the managed profile.
- `KEYGUARD_DISABLE_FINGERPRINT`, `KEYGUARD_DISABLE_FACE` or `KEYGUARD_DISABLE_IRIS` which affects the managed profile challenge if there is one, or the parent user otherwise.
- `KEYGUARD_DISABLE_UNREDACTED_NOTIFICATIONS` which affects notifications generated by applications in the managed profile.

From version `Build.VERSION_CODES.VANILLA_ICE_CREAM`, the profile owner of a managed profile can also set `KEYGUARD_DISABLE_WIDGETS_ALL` which disables keyguard widgets for the managed profile.

From version `Build.VERSION_CODES.R` the profile owner of an organization-owned managed profile can set:

- `KEYGUARD_DISABLE_SECURE_CAMERA` which affects the parent user when called on the parent profile.
- `KEYGUARD_DISABLE_SECURE_NOTIFICATIONS` which affects the parent user when called on the parent profile.

Starting from version `Build.VERSION_CODES.VANILLA_ICE_CREAM` the profile owner of an organization-owned managed profile can set:

- `KEYGUARD_DISABLE_WIDGETS_ALL` which affects the parent user when called on the parent profile.

`KEYGUARD_DISABLE_TRUST_AGENTS`, `KEYGUARD_DISABLE_FINGERPRINT`, `KEYGUARD_DISABLE_FACE`, `KEYGUARD_DISABLE_IRIS`, `KEYGUARD_DISABLE_SECURE_CAMERA` and `KEYGUARD_DISABLE_SECURE_NOTIFICATIONS` can also be set on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to set restrictions on the parent profile. `KEYGUARD_DISABLE_SECURE_CAMERA` can only be set on the parent profile instance if the calling device admin is the profile owner of an organization-owned managed profile.

Requests to disable other features on a managed profile will be ignored.

The admin can check which features have been disabled by calling `getKeyguardDisabledFeatures(ComponentName)`

Parameters	
admin	ComponentName : Which DeviceAdminReceiver this request is associated with. Null if the caller is not a device admin. This value may be null .
which	int : The disabled features flag which can be either KEYGUARD_DISABLE_FEATURES_NONE (default), KEYGUARD_DISABLE_FEATURES_ALL , or a combination of KEYGUARD_DISABLE_WIDGETS_ALL , KEYGUARD_DISABLE_SECURE_CAMERA , KEYGUARD_DISABLE_SECURE_NOTIFICATIONS , KEYGUARD_DISABLE_TRUST_AGENTS , KEYGUARD_DISABLE_UNREDACTED_NOTIFICATIONS , KEYGUARD_DISABLE_FINGERPRINT , KEYGUARD_DISABLE_FACE , KEYGUARD_DISABLE_IRIS , KEYGUARD_DISABLE_SHORTCUTS_ALL .

Throws	
SecurityException	if admin is not an active administrator or does not use DeviceAdminInfo.USES_POLICY_DISABLE_KEYGUARD_FEATURES

setLocationEnabled

```
public void setLocationEnabled (ComponentName admin,
                boolean locationEnabled)
```

Called by device owners to set the user's global location setting.

Parameters	
admin	ComponentName : Which DeviceAdminReceiver this request is associated with. This value cannot be null .
locationEnabled	boolean : whether location should be enabled or disabled. **Note:** on automotive builds , calls to disable will be ignored.

Throws	
SecurityException	if admin is not a device owner.

setLockTaskFeatures

```
public void setLockTaskFeatures (ComponentName admin,
                int flags)
```

Sets which system features are enabled when the device runs in lock task mode. This method doesn't affect the features when lock task mode is inactive. Any system features not included in flags are implicitly disabled when calling this method. By default, only LOCK_TASK_FEATURE_GLOBAL_ACTIONS is enabled; all the other features are disabled. To disable the global actions dialog, call this method omitting LOCK_TASK_FEATURE_GLOBAL_ACTIONS .

This method can only be called by the device owner, a profile owner of an affiliated user or profile, or the profile owner when no device owner is set or holders of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_LOCK_TASK`. See `isAffiliatedUser()`. Any features set using this method are cleared if the user becomes unaffiliated.

Starting from `Build.VERSION_CODES.UPSIDE_DOWN_CAKE`, after the lock task features policy has been set, `PolicyUpdateReceiver.onPolicySetResult(Context,String,Bundle,TargetUser,PolicyUpdateResult)` will notify the admin on whether the policy was successfully set or not. This callback will contain:

- The policy identifier `DevicePolicyIdentifiers.LOCK_TASK_POLICY`
- The `TargetUser` that this policy relates to
- The `PolicyUpdateResult`, which will be `PolicyUpdateResult.RESULT_POLICY_SET` if the policy was successfully set or the reason the policy failed to be set (e.g. `PolicyUpdateResult.RESULT_FAILURE_CONFLICTING_ADMIN_POLICY`)

If there has been a change to the policy, `PolicyUpdateReceiver.onPolicyChanged(Context,String,Bundle,TargetUser,PolicyUpdateResult)` will notify the admin of this change. This callback will contain the same parameters as PolicyUpdateReceiver#onPolicySetResult and the `PolicyUpdateResult` will contain the reason why the policy changed.

Starting from `Build.VERSION_CODES.UPSIDE_DOWN_CAKE`, lock task features and lock task packages are bundled as one policy. A failure to apply one will result in a failure to apply the other.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null`.
`flags`	`int`: The system features enabled during lock task mode. Value is either `0` or a combination of the following: • `LOCK_TASK_FEATURE_NONE` • `LOCK_TASK_FEATURE_SYSTEM_INFO` • `LOCK_TASK_FEATURE_NOTIFICATIONS` • `LOCK_TASK_FEATURE_HOME` • `LOCK_TASK_FEATURE_OVERVIEW` • `LOCK_TASK_FEATURE_GLOBAL_ACTIONS` • `LOCK_TASK_FEATURE_KEYGUARD` • `LOCK_TASK_FEATURE_BLOCK_ACTIVITY_START_IN_TASK`
Throws	
`SecurityException`	if `admin` is not the device owner, the profile owner of an affiliated user or profile, or the profile owner when no device owner is set or holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_LOCK_TASK`.

setLockTaskPackages

```
public void setLockTaskPackages (ComponentName admin,
                String[] packages)
```

Sets which packages may enter lock task mode.

Any packages that share uid with an allowed package will also be allowed to activate lock task. From `Build.VERSION_CODES.M` removing packages from the lock task package list results in locked tasks belonging to those packages to be finished.

This function can only be called by the device owner, a profile owner of an affiliated user or profile, or the profile owner when no device owner is set or holders of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_LOCK_TASK`. See `isAffiliatedUser()`. Any package set via this method will be cleared if the user becomes unaffiliated.

Starting from `Build.VERSION_CODES.UPSIDE_DOWN_CAKE`, after the lock task policy has been set, `PolicyUpdateReceiver.onPolicySetResult(Context,String,Bundle,TargetUser,PolicyUpdateResult)` will notify the admin on whether the policy was successfully set or not. This callback will contain:

- The policy identifier `DevicePolicyIdentifiers.LOCK_TASK_POLICY`
- The `TargetUser` that this policy relates to
- The `PolicyUpdateResult`, which will be `PolicyUpdateResult.RESULT_POLICY_SET` if the policy was successfully set or the reason the policy failed to be set (e.g. `PolicyUpdateResult.RESULT_FAILURE_CONFLICTING_ADMIN_POLICY`)

If there has been a change to the policy, `PolicyUpdateReceiver.onPolicyChanged(Context,String,Bundle,TargetUser,PolicyUpdateResult)` will notify the admin of this change. This callback will contain the same parameters as PolicyUpdateReceiver#onPolicySetResult and the `PolicyUpdateResult` will contain the reason why the policy changed.

Starting from `Build.VERSION_CODES.UPSIDE_DOWN_CAKE`, lock task features and lock task packages are bundled as one policy. A failure to apply one will result in a failure to apply the other.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null`.
`packages`	`String`: The list of packages allowed to enter lock task mode. This value cannot be `null`.
Throws	
`SecurityException`	if `admin` is not the device owner, the profile owner of an affiliated user or profile, or the profile owner when no device owner is set or holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_LOCK_TASK`.

See also:

- `isAffiliatedUser()`
- `Activity.startLockTask()`
- `DeviceAdminReceiver.onLockTaskModeEntering(Context,Intent,String)`
- `DeviceAdminReceiver.onLockTaskModeExiting(Context,Intent)`
- `UserManager.DISALLOW_CREATE_WINDOWS`

setLogoutEnabled

```
public void setLogoutEnabled (ComponentName admin,
                boolean enabled)
```

Called by a device owner to specify whether logout is enabled for all secondary users. The system may show a logout button that stops the user and switches back to the primary user.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
`enabled`	`boolean` : whether logout should be enabled or not.
Throws	
`SecurityException`	if `admin` is not a device owner.

setLongSupportMessage

```
public void setLongSupportMessage (ComponentName admin,
                CharSequence message)
```

Called by a device admin to set the long support message. This will be displayed to the user in the device administrators settings screen. If the message is longer than 20000 characters it may be truncated.

If the long support message needs to be localized, it is the responsibility of the `DeviceAdminReceiver` to listen to the `Intent.ACTION_LOCALE_CHANGED` broadcast and set a new version of this string accordingly.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
`message`	`CharSequence` : Long message to be displayed to the user in settings or null to clear the existing message.
Throws	
`SecurityException`	if `admin` is not an active administrator.

setManagedProfileCallerIdAccessPolicy

```
public void setManagedProfileCallerIdAccessPolicy (PackagePolicy policy)
```

Called by a profile owner of a managed profile to set the packages that are allowed to lookup contacts in the managed profile based on caller id information.

For example, the policy determines if a dialer app in the parent profile resolving an incoming call can search the caller id data, such as phone number, of managed contacts and return managed contacts that match.

The calling device admin must be a profile owner of a managed profile. If it is not, a `SecurityException` will be thrown.

A `PackagePolicy.PACKAGE_POLICY_ALLOWLIST_AND_SYSTEM` policy type allows access from the OEM default packages for the Sms, Dialer and Contact roles, in addition to the packages specified in `PackagePolicy.getPackageNames()`

Parameters	
`policy`	`PackagePolicy`: the policy to set, setting this value to `null` will allow all packages
Throws	
`SecurityException`	if caller is not a profile owner of a managed profile

setManagedProfileContactsAccessPolicy

```
public void setManagedProfileContactsAccessPolicy (PackagePolicy policy)
```

Called by a profile owner of a managed profile to set the packages that are allowed access to the managed profile contacts from the parent user.

For example, the system will enforce the provided policy and determine if contacts in the managed profile are shown when queried by an application in the parent user.

The calling device admin must be a profile owner of a managed profile. If it is not, a `SecurityException` will be thrown.

A `PackagePolicy.PACKAGE_POLICY_ALLOWLIST_AND_SYSTEM` policy type allows access from the OEM default packages for the Sms, Dialer and Contact roles, in addition to the packages specified in `PackagePolicy.getPackageNames()`

Parameters	
`policy`	`PackagePolicy`: the policy to set, setting this value to `null` will allow all packages
Throws	
`SecurityException`	if caller is not a profile owner of a managed profile

setManagedProfileMaximumTimeOff

```
public void setManagedProfileMaximumTimeOff (ComponentName admin,
                long timeoutMillis)
```

Called by a profile owner of an organization-owned managed profile to set maximum time the profile is allowed to be turned off. If the profile is turned off for longer, personal apps are suspended on the device.

When personal apps are suspended, an ongoing notification about that is shown to the user. When the user taps the notification, system invokes `ACTION_CHECK_POLICY_COMPLIANCE` in the profile owner package. Profile owner implementation that uses personal apps suspension must handle this intent.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
`timeoutMillis`	`long` : Maximum time the profile is allowed to be off in milliseconds or 0 if not limited. The minimum non-zero value corresponds to 72 hours. If an admin sets a smaller non-zero value, 72 hours will be set instead.

Throws	
`IllegalStateException`	if the profile owner doesn't have an activity that handles `ACTION_CHECK_POLICY_COMPLIANCE`

setManagedSubscriptionsPolicy

```
public void setManagedSubscriptionsPolicy (ManagedSubscriptionsPolicy policy)
```

Called by a profile owner of an organization-owned device to specify `ManagedSubscriptionsPolicy`

Managed subscriptions policy controls how SIMs would be associated with the managed profile. For example a policy of type `ManagedSubscriptionsPolicy.TYPE_ALL_MANAGED_SUBSCRIPTIONS` assigns all SIM-based subscriptions to the managed profile. In this case OEM default dialer and messages app are automatically installed in the managed profile and all incoming and outgoing calls and text messages are handled by them.

This API can only be called during device setup.

Parameters	
`policy`	`ManagedSubscriptionsPolicy` : `ManagedSubscriptionsPolicy` policy, passing null for this resets the policy to be the default.

Throws	
`IllegalStateException`	if called after the device setup has been completed.

`SecurityException`	if the caller is not a profile owner on an organization-owned managed profile.
`UnsupportedOperationException`	if managed subscriptions policy is not explicitly enabled by the device policy management role holder during device setup.

setMasterVolumeMuted

```
public void setMasterVolumeMuted (ComponentName admin,
                boolean on)
```

Called by profile or device owners to set the global volume mute on or off. This has no effect when set on a managed profile.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
`on`	`boolean` : `true` to mute global volume, `false` to turn mute off.

Throws	
`SecurityException`	if `admin` is not a device or profile owner.

setMaximumFailedPasswordsForWipe

```
public void setMaximumFailedPasswordsForWipe (ComponentName admin,
                int num)
```

Setting this to a value greater than zero enables a policy that will perform a device or profile wipe after too many incorrect device-unlock passwords have been entered. This policy combines watching for failed passwords and wiping the device, and requires that calling Device Admins request both `DeviceAdminInfo.USES_POLICY_WATCH_LOGIN` and `DeviceAdminInfo.USES_POLICY_WIPE_DATA` }.

When this policy is set on the system or the main user, the device will be factory reset after too many incorrect password attempts. When set on any other user, only the corresponding user or profile will be wiped.

To implement any other policy (e.g. wiping data for a particular application only, erasing or revoking credentials, or reporting the failure to a server), you should implement `DeviceAdminReceiver.onPasswordFailed(Context,android.content.Intent)` instead. Do not use this API, because if the maximum count is reached, the device or profile will be wiped immediately, and your callback will not be invoked.

This method can be called on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to set a value on the parent profile. This allows a profile wipe after too many incorrect device-unlock password have been entered on the parent profile even if each profile has a separate challenge.

On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, the password is always empty and this method has no effect - i.e. the policy is not set.
Requires the `PackageManager#FEATURE_SECURE_LOCK_SCREEN` feature which can be detected using `PackageManager.hasSystemFeature(String)` .

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null` .
`num`	`int` : The number of failed password attempts at which point the device or profile will be wiped.
Throws	
`SecurityException`	if `admin` is not null, and `admin` is not an active administrator or does not use both `DeviceAdminInfo.USES_POLICY_WATCH_LOGIN` and `DeviceAdminInfo.USES_POLICY_WIPE_DATA` , or if `admin` is null and the caller does not have permission to wipe the device.

setMaximumTimeToLock

```
public void setMaximumTimeToLock (ComponentName admin,
                long timeMs)
```

Called by an application that is administering the device to set the maximum time for user activity until the device will lock. This limits the length that the user can set. It takes effect immediately.

A calling device admin must have requested `DeviceAdminInfo.USES_POLICY_FORCE_LOCK` to be able to call this method; if it has not, a security exception will be thrown.

This method can be called on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to set restrictions on the parent profile.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null` .
`timeMs`	`long` : The new desired maximum time to lock in milliseconds. A value of 0 means there is no restriction.
Throws	
`SecurityException`	if `admin` is not an active administrator or it does not use `DeviceAdminInfo.USES_POLICY_FORCE_LOCK`

setMeteredDataDisabledPackages

```
public List<String> setMeteredDataDisabledPackages (ComponentName admin,
                List<String> packageNames)
```

Called by a device or profile owner to restrict packages from using metered data.

Parameters	
`admin`	`ComponentName` : which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
`packageNames`	`List` : the list of package names to be restricted. This value cannot be `null` .
Returns	
`List<String>`	a list of package names which could not be restricted. This value cannot be `null` .
Throws	
`SecurityException`	if `admin` is not a device or profile owner.

setMinimumRequiredWifiSecurityLevel

```
public void setMinimumRequiredWifiSecurityLevel (int level)
```

Called by device owner or profile owner of an organization-owned managed profile to specify the minimum security level required for Wi-Fi networks. The device may not connect to networks that do not meet the minimum security level. If the current network does not meet the minimum security level set, it will be disconnected. The following shows the Wi-Fi security levels from the lowest to the highest security level: `WIFI_SECURITY_OPEN` `WIFI_SECURITY_PERSONAL` `WIFI_SECURITY_ENTERPRISE_EAP` `WIFI_SECURITY_ENTERPRISE_192`

Parameters	
`level`	`int` : minimum security level. Value is one of the following: • `WIFI_SECURITY_OPEN` • `WIFI_SECURITY_PERSONAL` • `WIFI_SECURITY_ENTERPRISE_EAP` • `WIFI_SECURITY_ENTERPRISE_192`
Throws	
`SecurityException`	if the caller is not permitted to set this policy

setMtePolicy

```
public void setMtePolicy (int policy)
```

Called by a device owner, profile owner of an organization-owned device, to set the Memory Tagging Extension (MTE) policy. MTE is a CPU extension that allows to protect against certain classes of security problems at a small runtime performance cost overhead.

The MTE policy can only be set to `MTE_DISABLED` if called by a device owner. Otherwise a `SecurityException` will be thrown.

The device needs to be rebooted to apply changes to the MTE policy.

Parameters	
`policy`	`int` : the MTE policy to be set. Value is one of the following: • `MTE_ENABLED` • `MTE_DISABLED` • `MTE_NOT_CONTROLLED_BY_POLICY`

Throws	
`SecurityException`	if caller is not permitted to set Mte policy
`UnsupportedOperationException`	if the device does not support MTE

setNearbyAppStreamingPolicy

```
public void setNearbyAppStreamingPolicy (int policy)
```

Called by a device/profile owner to set nearby app streaming policy. App streaming is when the device starts an app on a virtual display and sends a video stream of the app to nearby devices.

Parameters	
`policy`	`int` : One of the `NearbyStreamingPolicy` constants. Value is one of the following: • `NEARBY_STREAMING_NOT_CONTROLLED_BY_POLICY` • `NEARBY_STREAMING_DISABLED` • `NEARBY_STREAMING_ENABLED` • `NEARBY_STREAMING_SAME_MANAGED_ACCOUNT_ONLY`

Throws	
`SecurityException`	if caller is not a device or profile owner.

setNearbyNotificationStreamingPolicy

```
public void setNearbyNotificationStreamingPolicy (int policy)
```

Called by a device/profile owner to set nearby notification streaming policy. Notification streaming is sending notification data from pre-installed apps to nearby devices.

Parameters	
`policy`	`int` : One of the `NearbyStreamingPolicy` constants. Value is one of the following: • `NEARBY_STREAMING_NOT_CONTROLLED_BY_POLICY` • `NEARBY_STREAMING_DISABLED` • `NEARBY_STREAMING_ENABLED` • `NEARBY_STREAMING_SAME_MANAGED_ACCOUNT_ONLY`

Throws	
`SecurityException`	if caller is not a device or profile owner

setNetworkLoggingEnabled

```
public void setNetworkLoggingEnabled (ComponentName admin,
                boolean enabled)
```

Called by a device owner, profile owner of a managed profile or delegated app with `DELEGATION_NETWORK_LOGGING` to control the network logging feature.

Supported for a device owner from Android 8 and a delegated app granted by a device owner from Android 10. Supported for a profile owner of a managed profile and a delegated app granted by a profile owner from Android 12. When network logging is enabled by a profile owner, the network logs will only include work profile network activity, not activity on the personal profile.

Network logs contain DNS lookup and connect() library call events. The following library functions are recorded while network logging is active:

- `getaddrinfo()`
- `gethostbyname()`
- `connect()`

Network logging is a low-overhead tool for forensics but it is not guaranteed to use full system call logging; event reporting is enabled by default for all processes but not strongly enforced. Events from applications using alternative implementations of libc, making direct kernel calls, or deliberately obfuscating traffic may not be recorded.

Some common network events may not be reported. For example:

- Applications may hardcode IP addresses to reduce the number of DNS lookups, or use an alternative system for name resolution, and so avoid calling `getaddrinfo()` or `gethostbyname`.
- Applications may use datagram sockets for performance reasons, for example for a game client. Calling `connect()` is unnecessary for this kind of socket, so it will not trigger a network event.

It is possible to directly intercept layer 3 traffic leaving the device using an always-on VPN service. See `setAlwaysOnVpnPackage(ComponentName,String,boolean)` and `VpnService` for details.

Note: The device owner won't be able to retrieve network logs if there are unaffiliated secondary users or profiles on the device, regardless of whether the feature is enabled. Logs will be discarded if the internal buffer fills up while waiting for all users to become affiliated. Therefore it's recommended that affiliation ids are set for new users as soon as possible after provisioning via `setAffiliationIds(ComponentName, Set)`.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with, or `null` if called by a delegated app.
`enabled`	`boolean`: whether network logging should be enabled or not.

Throws	
`SecurityException`	if `admin` is not a device owner or profile owner.

setOrganizationColor

```
public void setOrganizationColor (ComponentName admin,
                int color)
```

This method was deprecated in API level 31.
From `Build.VERSION_CODES.R`, the organization color is never used as the background color of the confirm credentials screen.

Called by a profile owner of a managed profile to set the color used for customization. This color is used as background color of the confirm credentials screen for that user. The default color is teal (#00796B).

The confirm credentials screen can be created using `KeyguardManager.createConfirmDeviceCredentialIntent(CharSequence, CharSequence)`.

Starting from Android R, the organization color will no longer be used as the background color of the confirm credentials screen.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null`.
`color`	`int`: The 24bit (0xRRGGBB) representation of the color to be used.

Throws	
SecurityException	if admin is not a profile owner.

setOrganizationId

```
public void setOrganizationId (String enterpriseId)
```

Sets the Enterprise ID for the work profile or managed device. This is a requirement for generating an enrollment-specific ID for the device, see getEnrollmentSpecificId() . It is recommended that the Enterprise ID is at least 6 characters long, and no more than 64 characters.

Parameters	
enterpriseId	String : An identifier of the organization this work profile or device is enrolled into. This value cannot be null .

setOrganizationName

```
public void setOrganizationName (ComponentName admin,
                CharSequence title)
```

Called by the device owner (since API 26) or profile owner (since API 24) to set the name of the organization under management.

If the organization name needs to be localized, it is the responsibility of the caller to listen to the Intent.ACTION_LOCALE_CHANGED broadcast and set a new version of this string accordingly.

Parameters	
admin	ComponentName : Which DeviceAdminReceiver this request is associated with. Null if the caller is not a device admin. This value may be null .
title	CharSequence : The organization name or null to clear a previously set name.

Throws	
SecurityException	if admin is not a device or profile owner.

setOverrideApnsEnabled

```
public void setOverrideApnsEnabled (ComponentName admin,
                boolean enabled)
```

Called by device owner to set if override APNs should be enabled.

Override APNs are separated from other APNs on the device, and can only be inserted or modified by the device owner. When enabled, only override APNs are in use, any other APNs are ignored.

Note: Enterprise APNs added by managed profile owners do not need to be enabled by this API. They are part of the preferential network service config and is controlled by `setPreferentialNetworkServiceConfigs(List)` .

Parameters	
`admin`	`ComponentName` : which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
`enabled`	`boolean` : `true` if override APNs should be enabled, `false` otherwise
Throws	
`SecurityException`	if `admin` is not a device owner.

setPackagesSuspended

```
public String[] setPackagesSuspended (ComponentName admin,
                String[] packageNames,
                boolean suspended)
```

Called by device or profile owners to suspend packages for this user. This function can be called by a device owner, profile owner, or by a delegate given the `DELEGATION_PACKAGE_ACCESS` scope via `setDelegatedScopes(ComponentName, String, List)` .

A suspended package will not be able to start activities. Its notifications will be hidden, it will not show up in recents, will not be able to show toasts or dialogs or ring the device.

The package must already be installed. If the package is uninstalled while suspended the package will no longer be suspended. The admin can block this by using `setUninstallBlocked(ComponentName, String, boolean)` .

Some apps cannot be suspended, such as device admins, the active launcher, the required package installer, the required package uninstaller, the required package verifier, the default dialer, and the permission controller.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null` .
`packageNames`	`String` : The package names to suspend or unsuspend. This value cannot be `null` .
`suspended`	`boolean` : If set to `true` than the packages will be suspended, if set to `false` the packages will be unsuspended.
Returns	

`String[]`	an array of package names for which the suspended status is not set as requested in this method. This value cannot be `null`.

Throws	
`SecurityException`	if `admin` is not a device or profile owner.

setPasswordExpirationTimeout

```
public void setPasswordExpirationTimeout (ComponentName admin,
                long timeout)
```

Called by a device admin to set the password expiration timeout. Calling this method will restart the countdown for password expiration for the given admin, as will changing the device password (for all admins).

The provided timeout is the time delta in ms and will be added to the current time. For example, to have the password expire 5 days from now, timeout would be 5 * 86400 * 1000 = 432000000 ms for timeout.

To disable password expiration, a value of 0 may be used for timeout.

On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, the password expiration is always disabled.

A calling device admin must have requested `DeviceAdminInfo.USES_POLICY_EXPIRE_PASSWORD` to be able to call this method; if it has not, a security exception will be thrown.

Note that setting the password will automatically reset the expiration time for all active admins. Active admins do not need to explicitly call this method in that case.

This method can be called on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to set restrictions on the parent profile.
Requires the `PackageManager#FEATURE_SECURE_LOCK_SCREEN` feature which can be detected using `PackageManager.hasSystemFeature(String)`.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null`.
`timeout`	`long`: The limit (in ms) that a password can remain in effect. A value of 0 means there is no restriction (unlimited).

Throws	
`SecurityException`	if `admin` is not an active administrator or `admin` does not use `DeviceAdminInfo.USES_POLICY_EXPIRE_PASSWORD`

setPasswordHistoryLength

```
public void setPasswordHistoryLength (ComponentName admin,
                int length)
```

Called by an application that is administering the device to set the length of the password history. After setting this, the user will not be able to enter a new password that is the same as any password in the history. Note that the current password will remain until the user has set a new one, so the change does not take place immediately. To prompt the user for a new password, use `ACTION_SET_NEW_PASSWORD` or `ACTION_SET_NEW_PARENT_PROFILE_PASSWORD` after setting this value.

On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, the password history length is always 0.

The calling device admin must have requested `DeviceAdminInfo.USES_POLICY_LIMIT_PASSWORD` to be able to call this method; if it has not, a security exception will be thrown.

This method can be called on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to set restrictions on the parent profile.
Requires the `PackageManager#FEATURE_SECURE_LOCK_SCREEN` feature which can be detected using `PackageManager.hasSystemFeature(String)` .

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
`length`	`int` : The new desired length of password history. A value of 0 means there is no restriction.
Throws	
`SecurityException`	if `admin` is not an active administrator or `admin` does not use `DeviceAdminInfo.USES_POLICY_LIMIT_PASSWORD`

setPasswordMinimumLength

```
public void setPasswordMinimumLength (ComponentName admin,
                int length)
```

This method was deprecated in API level 31.
see `setPasswordQuality(ComponentName,int)` for details.

Called by an application that is administering the device to set the minimum allowed password length. After setting this, the user will not be able to enter a new password that is not at least as restrictive as what has been set. Note that the current password will remain until the user has set a new one, so the change does not take place immediately. To prompt the user for a new password, use `ACTION_SET_NEW_PASSWORD` or `ACTION_SET_NEW_PARENT_PROFILE_PASSWORD` after setting this value. This constraint is only imposed if the administrator has also requested either

`PASSWORD_QUALITY_NUMERIC` , `PASSWORD_QUALITY_NUMERIC_COMPLEX` , `PASSWORD_QUALITY_ALPHABETIC` , `PASSWORD_QUALITY_ALPHANUMERIC` , or `PASSWORD_QUALITY_COMPLEX` with `setPasswordQuality(ComponentName, int)` . If an app targeting SDK level `Build.VERSION_CODES.R` and above enforces this constraint without settings password quality to one of these values first, this method will throw `IllegalStateException` .

On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, the password is always treated as empty.

The calling device admin must have requested `DeviceAdminInfo.USES_POLICY_LIMIT_PASSWORD` to be able to call this method; if it has not, a security exception will be thrown.

Apps targeting `Build.VERSION_CODES.R` and below can call this method on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to set restrictions on the parent profile.

Note: this method is ignored on {PackageManager#FEATURE_AUTOMOTIVE automotive builds}.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
`length`	`int` : The new desired minimum password length. A value of 0 means there is no restriction.

Throws	
`IllegalStateException`	if the calling app is targeting SDK level `Build.VERSION_CODES.R` and above and didn't set a sufficient password quality requirement prior to calling this method.
`SecurityException`	if `admin` is not an active administrator or `admin` does not use `DeviceAdminInfo.USES_POLICY_LIMIT_PASSWORD`

setPasswordMinimumLetters

```
public void setPasswordMinimumLetters (ComponentName admin,
                int length)
```

This method was deprecated in API level 31.
see `setPasswordQuality(ComponentName,int)` for details.

Called by an application that is administering the device to set the minimum number of letters required in the password. After setting this, the user will not be able to enter a new password that is not at least as restrictive as what has been set. Note that the current password will remain until the user has set a new one, so the change does not take place immediately. To prompt the user for a new password, use `ACTION_SET_NEW_PASSWORD` or `ACTION_SET_NEW_PARENT_PROFILE_PASSWORD` after setting this value. This constraint is only imposed if the administrator has also requested `PASSWORD_QUALITY_COMPLEX` with `setPasswordQuality(ComponentName, int)` . If an app targeting SDK level `Build.VERSION_CODES.R` and above enforces this constraint without settings password quality to `PASSWORD_QUALITY_COMPLEX` first, this method will throw `IllegalStateException` . The default value is 1.

On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, the password is always treated as empty.

The calling device admin must have requested `DeviceAdminInfo.USES_POLICY_LIMIT_PASSWORD` to be able to call this method; if it has not, a security exception will be thrown.

Apps targeting `Build.VERSION_CODES.R` and below can call this method on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to set restrictions on the parent profile.

Note: this method is ignored on {PackageManager#FEATURE_AUTOMOTIVE automotive builds}.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
`length`	`int` : The new desired minimum number of letters required in the password. A value of 0 means there is no restriction.

Throws	
`IllegalStateException`	if the calling app is targeting SDK level `Build.VERSION_CODES.R` and above and didn't set a sufficient password quality requirement prior to calling this method.
`SecurityException`	if `admin` is not an active administrator or `admin` does not use `DeviceAdminInfo.USES_POLICY_LIMIT_PASSWORD`

setPasswordMinimumLowerCase

```
public void setPasswordMinimumLowerCase (ComponentName admin,
                int length)
```

This method was deprecated in API level 31.
see `setPasswordQuality(ComponentName,int)` for details.

Called by an application that is administering the device to set the minimum number of lower case letters required in the password. After setting this, the user will not be able to enter a new password that is not at least as restrictive as what has been set. Note that the current password will remain until the user has set a new one, so the change does not take place immediately. To prompt the user for a new password, use `ACTION_SET_NEW_PASSWORD` or `ACTION_SET_NEW_PARENT_PROFILE_PASSWORD` after setting this value. This constraint is only imposed if the administrator has also requested `PASSWORD_QUALITY_COMPLEX` with `setPasswordQuality(ComponentName, int)` . If an app targeting SDK level `Build.VERSION_CODES.R` and above enforces this constraint without settings password quality to `PASSWORD_QUALITY_COMPLEX` first, this method will throw `IllegalStateException` . The default value is 0.

On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, the password is always treated as empty.

The calling device admin must have requested `DeviceAdminInfo.USES_POLICY_LIMIT_PASSWORD` to be able to call this method; if it has not, a security exception will be thrown.

Apps targeting `Build.VERSION_CODES.R` and below can call this method on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to set restrictions on the parent profile.

Note: this method is ignored on {PackageManager#FEATURE_AUTOMOTIVE automotive builds}.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
`length`	`int` : The new desired minimum number of lower case letters required in the password. A value of 0 means there is no restriction.

Throws	
`IllegalStateException`	if the calling app is targeting SDK level `Build.VERSION_CODES.R` and above and didn't set a sufficient password quality requirement prior to calling this method.
`SecurityException`	if `admin` is not an active administrator or `admin` does not use `DeviceAdminInfo.USES_POLICY_LIMIT_PASSWORD`

setPasswordMinimumNonLetter

```
public void setPasswordMinimumNonLetter (ComponentName admin,
                int length)
```

This method was deprecated in API level 31.
see `setPasswordQuality(ComponentName,int)` for details.

Called by an application that is administering the device to set the minimum number of non-letter characters (numerical digits or symbols) required in the password. After setting this, the user will not be able to enter a new password that is not at least as restrictive as what has been set. Note that the current password will remain until the user has set a new one, so the change does not take place immediately. To prompt the user for a new password, use `ACTION_SET_NEW_PASSWORD` or `ACTION_SET_NEW_PARENT_PROFILE_PASSWORD` after setting this value. This constraint is only imposed if the administrator has also requested `PASSWORD_QUALITY_COMPLEX` with `setPasswordQuality(ComponentName, int)` . If an app targeting SDK level `Build.VERSION_CODES.R` and above enforces this constraint without settings password quality to `PASSWORD_QUALITY_COMPLEX` first, this method will throw `IllegalStateException` . The default value is 0.

On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, the password is always treated as empty.

The calling device admin must have requested `DeviceAdminInfo.USES_POLICY_LIMIT_PASSWORD` to be able to call this method; if it has not, a security exception will be thrown.

Apps targeting `Build.VERSION_CODES.R` and below can call this method on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to set restrictions on the parent profile.

Note: this method is ignored on {PackageManager#FEATURE_AUTOMOTIVE automotive builds}.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
`length`	`int` : The new desired minimum number of letters required in the password. A value of 0 means there is no restriction.
Throws	
`IllegalStateException`	if the calling app is targeting SDK level `Build.VERSION_CODES.R` and above and didn't set a sufficient password quality requirement prior to calling this method.
`SecurityException`	if `admin` is not an active administrator or `admin` does not use `DeviceAdminInfo.USES_POLICY_LIMIT_PASSWORD`

setPasswordMinimumNumeric

```
public void setPasswordMinimumNumeric (ComponentName admin,
                int length)
```

This method was deprecated in API level 31.
see `setPasswordQuality(ComponentName,int)` for details.

Called by an application that is administering the device to set the minimum number of numerical digits required in the password. After setting this, the user will not be able to enter a new password that is not at least as restrictive as what has been set. Note that the current password will remain until the user has set a new one, so the change does not take place immediately. To prompt the user for a new password, use `ACTION_SET_NEW_PASSWORD` or `ACTION_SET_NEW_PARENT_PROFILE_PASSWORD` after setting this value. This constraint is only imposed if the administrator has also requested `PASSWORD_QUALITY_COMPLEX` with `setPasswordQuality(ComponentName, int)` . If an app targeting SDK level `Build.VERSION_CODES.R` and above enforces this constraint without settings password quality to `PASSWORD_QUALITY_COMPLEX` first, this method will throw `IllegalStateException` . The default value is 1.

On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, the password is always treated as empty.

The calling device admin must have requested `DeviceAdminInfo.USES_POLICY_LIMIT_PASSWORD` to be able to call this method; if it has not, a security exception will be thrown.

Apps targeting `Build.VERSION_CODES.R` and below can call this method on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to set restrictions on the parent profile.

Note: this method is ignored on {PackageManager#FEATURE_AUTOMOTIVE automotive builds}.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
`length`	`int` : The new desired minimum number of numerical digits required in the password. A value of 0 means there is no restriction.

Throws	
`IllegalStateException`	if the calling app is targeting SDK level `Build.VERSION_CODES.R` and above and didn't set a sufficient password quality requirement prior to calling this method.
`SecurityException`	if `admin` is not an active administrator or `admin` does not use `DeviceAdminInfo.USES_POLICY_LIMIT_PASSWORD`

setPasswordMinimumSymbols

```
public void setPasswordMinimumSymbols (ComponentName admin,
                int length)
```

This method was deprecated in API level 31.
see `setPasswordQuality(ComponentName,int)` for details.

Called by an application that is administering the device to set the minimum number of symbols required in the password. After setting this, the user will not be able to enter a new password that is not at least as restrictive as what has been set. Note that the current password will remain until the user has set a new one, so the change does not take place immediately. To prompt the user for a new password, use `ACTION_SET_NEW_PASSWORD` or `ACTION_SET_NEW_PARENT_PROFILE_PASSWORD` after setting this value. This constraint is only imposed if the administrator has also requested `PASSWORD_QUALITY_COMPLEX` with `setPasswordQuality(ComponentName, int)` . If an app targeting SDK level `Build.VERSION_CODES.R` and above enforces this constraint without settings password quality to `PASSWORD_QUALITY_COMPLEX` first, this method will throw `IllegalStateException` . The default value is 1.

On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, the password is always treated as empty.

The calling device admin must have requested `DeviceAdminInfo.USES_POLICY_LIMIT_PASSWORD` to be able to call this method; if it has not, a security exception will be thrown.

Apps targeting `Build.VERSION_CODES.R` and below can call this method on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to set restrictions on the parent profile.

Note: this method is ignored on {PackageManager#FEATURE_AUTOMOTIVE automotive builds}.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .

`length`	`int` : The new desired minimum number of symbols required in the password. A value of 0 means there is no restriction.
Throws	
`IllegalStateException`	if the calling app is targeting SDK level `Build.VERSION_CODES.R` and above and didn't set a sufficient password quality requirement prior to calling this method.
`SecurityException`	if `admin` is not an active administrator or `admin` does not use `DeviceAdminInfo.USES_POLICY_LIMIT_PASSWORD`

setPasswordMinimumUpperCase

```
public void setPasswordMinimumUpperCase (ComponentName admin,
                int length)
```

This method was deprecated in API level 31.
see `setPasswordQuality(ComponentName,int)` for details.

Called by an application that is administering the device to set the minimum number of upper case letters required in the password. After setting this, the user will not be able to enter a new password that is not at least as restrictive as what has been set. Note that the current password will remain until the user has set a new one, so the change does not take place immediately. To prompt the user for a new password, use `ACTION_SET_NEW_PASSWORD` or `ACTION_SET_NEW_PARENT_PROFILE_PASSWORD` after setting this value. This constraint is only imposed if the administrator has also requested `PASSWORD_QUALITY_COMPLEX` with `setPasswordQuality(ComponentName, int)`. If an app targeting SDK level `Build.VERSION_CODES.R` and above enforces this constraint without settings password quality to `PASSWORD_QUALITY_COMPLEX` first, this method will throw `IllegalStateException`. The default value is 0.

On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, the password is always treated as empty.

The calling device admin must have requested `DeviceAdminInfo.USES_POLICY_LIMIT_PASSWORD` to be able to call this method; if it has not, a security exception will be thrown.

Apps targeting `Build.VERSION_CODES.R` and below can call this method on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to set restrictions on the parent profile.

Note: this method is ignored on {PackageManager#FEATURE_AUTOMOTIVE automotive builds}.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null`.
`length`	`int` : The new desired minimum number of upper case letters required in the password. A value of 0 means there is no restriction.
Throws	

`IllegalStateException`	if the calling app is targeting SDK level `Build.VERSION_CODES.R` and above and didn't set a sufficient password quality requirement prior to calling this method.
`SecurityException`	if `admin` is not an active administrator or `admin` does not use `DeviceAdminInfo.USES_POLICY_LIMIT_PASSWORD`

setPasswordQuality

```
public void setPasswordQuality (ComponentName admin,
                int quality)
```

This method was deprecated in API level 31.
Prefer using `setRequiredPasswordComplexity(int)`, to require a password that satisfies a complexity level defined by the platform, rather than specifying custom password requirement. Setting custom, overly-complicated password requirements leads to passwords that are hard for users to remember and may not provide any security benefits given as Android uses hardware-backed throttling to thwart online and offline brute-forcing of the device's screen lock. Company-owned devices (fully-managed and organization-owned managed profile devices) are able to continue using this method, though it is recommended that `setRequiredPasswordComplexity(int)` should be used instead.

Called by an application that is administering the device to set the password restrictions it is imposing. After setting this, the user will not be able to enter a new password that is not at least as restrictive as what has been set. Note that the current password will remain until the user has set a new one, so the change does not take place immediately. To prompt the user for a new password, use `ACTION_SET_NEW_PASSWORD` or `ACTION_SET_NEW_PARENT_PROFILE_PASSWORD` after calling this method.

Quality constants are ordered so that higher values are more restrictive; thus the highest requested quality constant (between the policy set here, the user's preference, and any other considerations) is the one that is in effect.

On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, the password is always treated as empty.

The calling device admin must have requested `DeviceAdminInfo.USES_POLICY_LIMIT_PASSWORD` to be able to call this method; if it has not, a security exception will be thrown.

Apps targeting `Build.VERSION_CODES.R` and below can call this method on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to set restrictions on the parent profile. Apps targeting `Build.VERSION_CODES.S` and above, with the exception of a profile owner on an organization-owned device (as can be identified by `isOrganizationOwnedDeviceWithManagedProfile()`), will get a `IllegalArgumentException` when calling this method on the parent `DevicePolicyManager` instance.

Note: Specifying password requirements using this method clears the password complexity requirements set using `setRequiredPasswordComplexity(int)`. If this method is called on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)`, then password complexity requirements set on the primary `DevicePolicyManager` must be cleared first by calling `setRequiredPasswordComplexity(int)` with `PASSWORD_COMPLEXITY_NONE` first.

Note: this method is ignored on {PackageManager#FEATURE_AUTOMOTIVE automotive builds}.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
`quality`	`int` : The new desired quality. One of `PASSWORD_QUALITY_UNSPECIFIED` , `PASSWORD_QUALITY_BIOMETRIC_WEAK` , `PASSWORD_QUALITY_SOMETHING` , `PASSWORD_QUALITY_NUMERIC` , `PASSWORD_QUALITY_NUMERIC_COMPLEX` , `PASSWORD_QUALITY_ALPHABETIC` , `PASSWORD_QUALITY_ALPHANUMERIC` or `PASSWORD_QUALITY_COMPLEX` .
Throws	
`IllegalStateException`	if the caller is trying to set password quality on the parent `DevicePolicyManager` instance while password complexity was set on the primary `DevicePolicyManager` instance.
`SecurityException`	if `admin` is not an active administrator, if `admin` does not use `DeviceAdminInfo.USES_POLICY_LIMIT_PASSWORD` or if the calling app is targeting `Build.VERSION_CODES.S` and above, and is calling the method the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` .

setPermissionGrantState

```
public boolean setPermissionGrantState (ComponentName admin,
                String packageName,
                String permission,
                int grantState)
```

Sets the grant state of a runtime permission for a specific application. The state can be `default` in which a user can manage it through the UI, `denied` , in which the permission is denied and the user cannot manage it through the UI, and `granted` in which the permission is granted and the user cannot manage it through the UI. This method can only be called by a profile owner, device owner, or a delegate given the `DELEGATION_PERMISSION_GRANT` scope via `setDelegatedScopes(ComponentName, String, List)` .

Note that user cannot manage other permissions in the affected group through the UI either and their granted state will be kept as the current value. Thus, it's recommended that you set the grant state of all the permissions in the affected group.

Setting the grant state to `default` does not revoke the permission. It retains the previous grant, if any.

Device admins with a `targetSdkVersion` < `Build.VERSION_CODES.Q` cannot grant and revoke permissions for applications built with a `targetSdkVersion` < `Build.VERSION_CODES.M` .

Admins with a `targetSdkVersion` ≥ `Build.VERSION_CODES.Q` can grant and revoke permissions of all apps. Similar to the user revoking a permission from a application built with a `targetSdkVersion` < `Build.VERSION_CODES.M` the app-op matching the permission is set to `AppOpsManager.MODE_IGNORED` , but the permission stays granted.

NOTE: On devices running `Build.VERSION_CODES.S` and above, control over the following, sensors-related, permissions is restricted:

- Manifest.permission.ACCESS_FINE_LOCATION
- Manifest.permission.ACCESS_BACKGROUND_LOCATION
- Manifest.permission.ACCESS_COARSE_LOCATION
- Manifest.permission.CAMERA
- Manifest.permission.RECORD_AUDIO
- Manifest.permission.RECORD_BACKGROUND_AUDIO
- Manifest.permission.ACTIVITY_RECOGNITION
- Manifest.permission.BODY_SENSORS

On devices running `Build.VERSION_CODES.BAKLAVA`, the `HealthPermissions` are also included in the restricted list.

A profile owner may not grant these permissions (i.e. call this method with any of the permissions listed above and `grantState` of `#PERMISSION_GRANT_STATE_GRANTED`), but may deny them.

A device owner, by default, may continue granting these permissions. However, for increased user control, the admin may opt out of controlling grants for these permissions by including `EXTRA_PROVISIONING_SENSORS_PERMISSION_GRANT_OPT_OUT` in the provisioning parameters. In that case the device owner's control will be limited to denying these permissions.

When sensor-related permissions aren't grantable due to the above cases, calling this method to grant these permissions will silently fail, if device admins are built with `targetSdkVersion` < `Build.VERSION_CODES.VANILLA_ICE_CREAM`. If they are built with `targetSdkVersion` >= `Build.VERSION_CODES.VANILLA_ICE_CREAM`, this method will throw a `SecurityException`.

NOTE: On devices running `Build.VERSION_CODES.S` and above, control over the following permissions are restricted for managed profile owners:

- Manifest.permission.READ_SMS

A managed profile owner may not grant these permissions (i.e. call this method with any of the permissions listed above and `grantState` of `#PERMISSION_GRANT_STATE_GRANTED`), but may deny them.

Attempts by the admin to grant these permissions, when the admin is restricted from doing so, will be silently ignored (no exception will be thrown). Control over the following permissions are restricted for managed profile owners:

- Manifest.permission.READ_SMS

A managed profile owner may not grant these permissions (i.e. call this method with any of the permissions listed above and `grantState` of `#PERMISSION_GRANT_STATE_GRANTED`), but may deny them.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null`.
`packageName`	`String`: The application to grant or revoke a permission to. This value cannot be `null`.

`permission`	`String` : The permission to grant or revoke. This value cannot be `null` .
`grantState`	`int` : The permission grant state which is one of `PERMISSION_GRANT_STATE_DENIED` , `PERMISSION_GRANT_STATE_DEFAULT` , `PERMISSION_GRANT_STATE_GRANTED` ,. Value is one of the following: • `PERMISSION_GRANT_STATE_DEFAULT` • `PERMISSION_GRANT_STATE_GRANTED` • `PERMISSION_GRANT_STATE_DENIED`

Returns	
`boolean`	whether the permission was successfully granted or revoked.

Throws	
`SecurityException`	if `admin` is not a device or profile owner.

See also:

- `PERMISSION_GRANT_STATE_DENIED`
- `PERMISSION_GRANT_STATE_DEFAULT`
- `PERMISSION_GRANT_STATE_GRANTED`
- `setDelegatedScopes(ComponentName, String, List)`
- `DELEGATION_PERMISSION_GRANT`

setPermissionPolicy

```
public void setPermissionPolicy (ComponentName admin,
                int policy)
```

Set the default response for future runtime permission requests by applications. This function can be called by a device owner, profile owner, or by a delegate given the `DELEGATION_PERMISSION_GRANT` scope via `setDelegatedScopes(ComponentName, String, List)` . The policy can allow for normal operation which prompts the user to grant a permission, or can allow automatic granting or denying of runtime permission requests by an application. This also applies to new permissions declared by app updates. When a permission is denied or granted this way, the effect is equivalent to setting the permission * grant state via `setPermissionGrantState(ComponentName, String, String, int)` .

As this policy only acts on runtime permission requests, it only applies to applications built with a `targetSdkVersion` of `Build.VERSION_CODES.M` or later.

NOTE: On devices running `Build.VERSION_CODES.S` and above, an auto-grant policy will not apply to certain sensors-related permissions on some configurations. See `setPermissionGrantState(ComponentName,String,String,int)` for the list of permissions affected, and the behavior change for managed profiles and fully-managed devices.

Parameters	
`admin`	`ComponentName` : Which profile or device owner this request is associated with, or `null` if the caller is a set permission policy delegate.
`policy`	`int` : One of the policy constants `PERMISSION_POLICY_PROMPT` , `PERMISSION_POLICY_AUTO_GRANT` and `PERMISSION_POLICY_AUTO_DENY` .

Throws	
`SecurityException`	if `admin` is not a device or profile owner.

See also:

- `setPermissionGrantState(ComponentName, String, String, int)`
- `setDelegatedScopes(ComponentName, String, List)`
- `DELEGATION_PERMISSION_GRANT`

setPermittedAccessibilityServices

```
public boolean setPermittedAccessibilityServices (ComponentName admin,
                List<String> packageNames)
```

Called by a profile or device owner to set the permitted `AccessibilityService` . When set by a device owner or profile owner the restriction applies to all profiles of the user the device owner or profile owner is an admin for. By default, the user can use any accessibility service. When zero or more packages have been added, accessibility services that are not in the list and not part of the system can not be enabled by the user.

Calling with a `null` value for the list disables the restriction so that all services can be used, calling with an empty list only allows the built-in system services. Any non-system accessibility service that's currently enabled must be included in the list.

System accessibility services are always available to the user and this method can't disable them.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
`packageNames`	`List` : List of accessibility service package names.

Returns	
`boolean`	`true` if the operation succeeded, or `false` if the list didn't contain every enabled non-system accessibility service.

Throws	
`SecurityException`	if `admin` is not a device or profile owner.

setPermittedCrossProfileNotificationListeners

```
public boolean setPermittedCrossProfileNotificationListeners (ComponentName admin,
                List<String> packageList)
```

Called by a profile owner of a managed profile to set the packages that are allowed to use a `NotificationListenerService` in the primary user to see notifications from the managed profile. By default all packages are permitted by this policy. When zero or more packages have been added, notification listeners installed on the primary user that are not in the list and are not part of the system won't receive events for managed profile notifications.

Calling with a `null` value for the list disables the restriction so that all notification listener services be used. Calling with an empty list disables all but the system's own notification listeners. System notification listener services are always available to the user.

If a device or profile owner want to stop notification listeners in their user from seeing that user's notifications they should prevent that service from running instead (e.g. via `setApplicationHidden(ComponentName,String,boolean)`)

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null`.
`packageList`	`List` : List of package names to allowlist. This value may be `null`.
Returns	
`boolean`	true if setting the restriction succeeded. It will fail if called outside a managed profile
Throws	
`SecurityException`	if `admin` is not a profile owner.

setPermittedInputMethods

```
public boolean setPermittedInputMethods (ComponentName admin,
                List<String> packageNames)
```

Called by a profile or device owner or holder of the `Manifest.permission.MANAGE_DEVICE_POLICY_INPUT_METHODS` permission to set the permitted input methods services for this user. By default, the user can use any input method.

This method can be called on the `DevicePolicyManager` instance, returned by `getParentProfileInstance(ComponentName)`, where the caller must be a profile owner of an organization-owned device.

If called on the parent instance:

- The permitted input methods will be applied on the personal profile
- Can only permit all input methods (calling this method with a `null` package list) or only permit system input methods (calling this method with an empty package list). This is to prevent the caller from learning which packages are installed on the personal side

When zero or more packages have been added, input method that are not in the list and not part of the system can not be enabled by the user. This method will fail if it is called for a admin that is not for the foreground user or a profile of the foreground user. Any non-system input method service that's currently enabled must be included in the list.

Calling with a null value for the list disables the restriction so that all input methods can be used, calling with an empty list disables all but the system's own input methods.

System input methods are always available to the user - this method can't modify this.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null` .
`packageNames`	`List` : List of input method package names.
Returns	
`boolean`	`true` if the operation succeeded, or `false` if the list didn't contain every enabled non-system input method service.
Throws	
`IllegalArgumentException`	if called on the parent profile, the `admin` is a profile owner of an organization-owned managed profile and the list of permitted input method package names is not null or empty.
`SecurityException`	if `admin` is not a device or profile owner and does not hold the `Manifest.permission.MANAGE_DEVICE_POLICY_INPUT_METHODS` permission, or if called on the parent profile and the `admin` is not a profile owner of an organization-owned managed profile.

setPersonalAppsSuspended

```
public void setPersonalAppsSuspended (ComponentName admin,
                boolean suspended)
```

Called by a profile owner of an organization-owned managed profile to suspend personal apps on the device. When personal apps are suspended the device can only be used for calls.

Parameters

`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
`suspended`	`boolean` : Whether personal apps should be suspended.
Throws	
`IllegalStateException`	if the profile owner doesn't have an activity that handles `ACTION_CHECK_POLICY_COMPLIANCE`

setPreferentialNetworkServiceConfigs

```
public void setPreferentialNetworkServiceConfigs (List<PreferentialNetworkServiceConfig> preferentialNetw
```

Sets preferential network configurations. An example of a supported preferential network service is the Enterprise slice on 5G networks. For devices on 4G networks, the profile owner needs to additionally configure enterprise APN to set up data call for the preferential network service. These APNs can be added using `addOverrideApn(ComponentName, ApnSetting)` . By default, preferential network service is disabled on the work profile and fully managed devices, on supported carriers and devices. Admins can explicitly enable it with this API. If admin wants to have multiple enterprise slices, it can be configured by passing list of `PreferentialNetworkServiceConfig` objects.

Parameters	
`preferentialNetworkServiceConfigs`	`List` : list of preferential network configurations. This value cannot be `null` .
Throws	
`SecurityException`	if the caller is not the profile owner or device owner.

setPreferentialNetworkServiceEnabled

```
public void setPreferentialNetworkServiceEnabled (boolean enabled)
```

Sets whether preferential network service is enabled. For example, an organization can have a deal/agreement with a carrier that all of the work data from its employees\u2019 devices will be sent via a network service dedicated for enterprise use. An example of a supported preferential network service is the Enterprise slice on 5G networks. For devices on 4G networks, the profile owner needs to additionally configure enterprise APN to set up data call for the preferential network service. These APNs can be added using `addOverrideApn(ComponentName, ApnSetting)` . By default, preferential network service is disabled on the work profile and fully managed devices, on supported carriers and devices. Admins can explicitly enable it with this API.

This method enables preferential network service with a default configuration. To fine-tune the configuration, use `setPreferentialNetworkServiceConfigs(List)` instead.

Before Android version `Build.VERSION_CODES.TIRAMISU` : this method can be called by the profile owner of a managed profile.

Starting from Android version `Build.VERSION_CODES.TIRAMISU` : This method can be called by the profile owner of a managed profile or device owner.

Parameters	
`enabled`	`boolean` : whether preferential network service should be enabled.
Throws	
`SecurityException`	if the caller is not the profile owner or device owner.

setProfileEnabled

```
public void setProfileEnabled (ComponentName admin)
```

Sets the enabled state of the profile. A profile should be enabled only once it is ready to be used. Only the profile owner can call this.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
Throws	
`SecurityException`	if `admin` is not a profile owner.

setProfileName

```
public void setProfileName (ComponentName admin,
                String profileName)
```

Sets the name of the profile. In the device owner case it sets the name of the user which it is called from. Only a profile owner or device owner can call this. If this is never called by the profile or device owner, the name will be set to default values.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associate with. This value cannot be `null` .
`profileName`	`String` : The name of the profile. If the name is longer than 200 characters it will be truncated.
Throws	

SecurityException	if admin is not a device or profile owner.

setRecommendedGlobalProxy

```
public void setRecommendedGlobalProxy (ComponentName admin,
                ProxyInfo proxyInfo)
```

Set a network-independent global HTTP proxy. This is not normally what you want for typical HTTP proxies - they are generally network dependent. However if you're doing something unusual like general internal filtering this may be useful. On a private network where the proxy is not accessible, you may break HTTP using this.

This method requires the caller to be the device owner.

This proxy is only a recommendation and it is possible that some apps will ignore it.

Note: The device owner won't be able to set a global HTTP proxy if there are unaffiliated secondary users or profiles on the device. It's recommended that affiliation ids are set for new users as soon as possible after provisioning via setAffiliationIds(ComponentName, Set) .

Parameters	
admin	ComponentName : Which DeviceAdminReceiver this request is associated with. This value cannot be null .
proxyInfo	ProxyInfo : The a ProxyInfo object defining the new global HTTP proxy. A null value will clear the global HTTP proxy.

Throws	
SecurityException	if admin is not the device owner.

setRequiredPasswordComplexity

```
public void setRequiredPasswordComplexity (int passwordComplexity)
```

Sets a minimum password complexity requirement for the user's screen lock. The complexity level is one of the pre-defined levels, and the user is unable to set a password with a lower complexity level.

Note that when called on a profile which uses an unified challenge with its parent, the complexity would apply to the unified challenge.

This method can be called on the DevicePolicyManager instance returned by getParentProfileInstance(ComponentName) in order to set restrictions on the parent profile.

Note: Specifying password requirements using this method clears any password requirements set using the obsolete setPasswordQuality(ComponentName,int) and any of its associated methods. Additionally, if there are password requirements set using the obsolete setPasswordQuality(ComponentName,int) on the parent DevicePolicyManager

instance, they must be cleared by calling `setPasswordQuality(ComponentName,int)` with `PASSWORD_QUALITY_UNSPECIFIED` on that instance prior to setting complexity requirement for the managed profile. Starting from `Build.VERSION_CODES.VANILLA_ICE_CREAM`, after the password requirement has been set, `PolicyUpdateReceiver.onPolicySetResult(Context,String,Bundle,TargetUser,PolicyUpdateResult)` will notify the admin on whether the policy was successfully set or not. This callback will contain:

- The policy identifier `DevicePolicyIdentifiers.PASSWORD_COMPLEXITY_POLICY`
- The `TargetUser` that this policy relates to
- The `PolicyUpdateResult`, which will be `PolicyUpdateResult.RESULT_POLICY_SET` if the policy was successfully set or the reason the policy failed to be set e.g. `PolicyUpdateResult.RESULT_FAILURE_CONFLICTING_ADMIN_POLICY`)

If there has been a change to the policy, `PolicyUpdateReceiver.onPolicyChanged(Context,String,Bundle,TargetUser,PolicyUpdateResult)` will notify the admin of this change. This callback will contain the same parameters as PolicyUpdateReceiver#onPolicySetResult and the `PolicyUpdateResult` will contain the reason why the policy changed.

Parameters	
`passwordComplexity`	`int`: Value is one of the following: • `PASSWORD_COMPLEXITY_NONE` • `PASSWORD_COMPLEXITY_LOW` • `PASSWORD_COMPLEXITY_MEDIUM` • `PASSWORD_COMPLEXITY_HIGH`

Throws	
`IllegalArgumentException`	if the complexity level is not one of the four above.
`IllegalStateException`	if the caller is trying to set password complexity while there are password requirements specified using `setPasswordQuality(ComponentName,int)` on the parent `DevicePolicyManager` instance.
`SecurityException`	if the calling application is not a device owner or a profile owner.

setRequiredStrongAuthTimeout

```
public void setRequiredStrongAuthTimeout (ComponentName admin,
                long timeoutMs)
```

Called by a device/profile owner to set the timeout after which unlocking with secondary, non strong auth (e.g. fingerprint, face, trust agents) times out, i.e. the user has to use a strong authentication method like password, pin or pattern.

This timeout is used internally to reset the timer to require strong auth again after specified timeout each time it has been successfully used.

Fingerprint can also be disabled altogether using `KEYGUARD_DISABLE_FINGERPRINT`.

Trust agents can also be disabled altogether using `KEYGUARD_DISABLE_TRUST_AGENTS`.

A calling device admin can verify the value it has set by calling `getRequiredStrongAuthTimeout(ComponentName)` and passing in its instance.

This method can be called on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to set restrictions on the parent profile.

On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, calling this methods has no effect - i.e. the timeout is not set.
Requires the `PackageManager#FEATURE_SECURE_LOCK_SCREEN` feature which can be detected using `PackageManager.hasSystemFeature(String)`.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null`.
`timeoutMs`	`long`: The new timeout in milliseconds, after which the user will have to unlock with strong authentication method. A value of 0 means the admin is not participating in controlling the timeout. The minimum and maximum timeouts are platform-defined and are typically 1 hour and 72 hours, respectively. Though discouraged, the admin may choose to require strong auth at all times using `KEYGUARD_DISABLE_FINGERPRINT` and/or `KEYGUARD_DISABLE_TRUST_AGENTS`.

Throws	
`SecurityException`	if `admin` is not a device or profile owner.

setResetPasswordToken

```
public boolean setResetPasswordToken (ComponentName admin,
                byte[] token)
```

Called by a profile or device owner to provision a token which can later be used to reset the device lockscreen password (if called by device owner), or managed profile challenge (if called by profile owner), via `resetPasswordWithToken(ComponentName, String, byte, int)`.

If the user currently has a lockscreen password, the provisioned token will not be immediately usable; it only becomes active after the user performs a confirm credential operation, which can be triggered by `KeyguardManager.createConfirmDeviceCredentialIntent`. If the user has no lockscreen password, the token is activated immediately. In all cases, the active state of the current token can be checked by

`isResetPasswordTokenActive(ComponentName)`. For security reasons, un-activated tokens are only stored in memory and will be lost once the device reboots. In this case a new token needs to be provisioned again.

Once provisioned and activated, the token will remain effective even if the user changes or clears the lockscreen password.

This token is highly sensitive and should be treated at the same level as user credentials. In particular, NEVER store this token on device in plaintext. Do not store the plaintext token in device-encrypted storage if it will be needed to reset password on file-based encryption devices before user unlocks. Consider carefully how any password token will be stored on your server and who will need access to them. Tokens may be the subject of legal access requests.

On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, the reset token is not set and this method returns false.
Requires the `PackageManager#FEATURE_SECURE_LOCK_SCREEN` feature which can be detected using `PackageManager.hasSystemFeature(String)`.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null`.
`token`	`byte`: a secure token a least 32-byte long, which must be generated by a cryptographically strong random number generator.

Returns	
`boolean`	true if the operation is successful, false otherwise.

Throws	
`IllegalArgumentException`	if the supplied token is invalid.
`SecurityException`	if admin is not a device or profile owner.

setRestrictionsProvider

```
public void setRestrictionsProvider (ComponentName admin,
                ComponentName provider)
```

Designates a specific service component as the provider for making permission requests of a local or remote administrator of the user.

Only a device owner or profile owner can designate the restrictions provider.

Parameters

`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
`provider`	`ComponentName` : The component name of the service that implements `RestrictionsReceiver` . If this param is null, it removes the restrictions provider previously assigned.

Throws	
`SecurityException`	if `admin` is not a device or profile owner.

setScreenCaptureDisabled

```
public void setScreenCaptureDisabled (ComponentName admin,
                boolean disabled)
```

Called by a device/profile owner to set whether the screen capture is disabled. Disabling screen capture also prevents the content from being shown on display devices that do not have a secure video output. See `Display.FLAG_SECURE` for more details about secure surfaces and secure displays.

This method can be called on the `DevicePolicyManager` instance, returned by `getParentProfileInstance(ComponentName)` , where the calling device admin must be the profile owner of an organization-owned managed profile. If it is not, a security exception will be thrown.

If the caller is device owner or called on the parent instance by a profile owner of an organization-owned managed profile, then the restriction will be applied to all users.

From version `Build.VERSION_CODES.M` disabling screen capture also blocks assist requests for all activities of the relevant user.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null` .
`disabled`	`boolean` : Whether screen capture is disabled or not.

Throws	
`SecurityException`	if the caller is not permitted to control screen capture policy.

setSecureSetting

```
public void setSecureSetting (ComponentName admin,
                String setting,
                String value)
```

This method is mostly deprecated. Most of the settings that still have an effect have dedicated setter methods (e.g. `setLocationEnabled(ComponentName, boolean)`) or user restrictions.

Called by profile or device owners to update `Settings.Secure` settings. Validation that the value of the setting is in the correct form for the setting type should be performed by the caller.

The settings that can be updated by a profile or device owner with this method are:

- `Settings.Secure.DEFAULT_INPUT_METHOD`
- `Settings.Secure.SKIP_FIRST_USE_HINTS`

A device owner can additionally update the following settings:

- `Settings.Secure.LOCATION_MODE`, but see note below.

Note: Starting from Android O, apps should no longer call this method with the setting `Settings.Secure.INSTALL_NON_MARKET_APPS`, which is deprecated. Instead, device owners or profile owners should use the restriction `UserManager.DISALLOW_INSTALL_UNKNOWN_SOURCES`. If any app targeting `Build.VERSION_CODES.O` or higher calls this method with `Settings.Secure.INSTALL_NON_MARKET_APPS`, an `UnsupportedOperationException` is thrown. Starting from Android Q, the device and profile owner can also call `UserManager.DISALLOW_INSTALL_UNKNOWN_SOURCES_GLOBALLY` to restrict unknown sources for all users. Note: Starting from Android R, apps should no longer call this method with the setting `Settings.Secure.LOCATION_MODE`, which is deprecated. Instead, device owners should call `setLocationEnabled(ComponentName,boolean)`. This will be enforced for all apps targeting Android R or above.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null`.
`setting`	`String`: The name of the setting to update.
`value`	`String`: The value to update the setting to.
Throws	
`SecurityException`	if `admin` is not a device or profile owner.

setSecurityLoggingEnabled

```
public void setSecurityLoggingEnabled (ComponentName admin,
                boolean enabled)
```

Called by device owner or a profile owner of an organization-owned managed profile to control the security logging feature.

Security logs contain various information intended for security auditing purposes. When security logging is enabled by any app other than the device owner, certain security logs are not visible (for example personal app launch events) or

they will be redacted (for example, details of the physical volume mount events). Please see `SecurityEvent` for details.

Note: The device owner won't be able to retrieve security logs if there are unaffiliated secondary users or profiles on the device, regardless of whether the feature is enabled. Logs will be discarded if the internal buffer fills up while waiting for all users to become affiliated. Therefore it's recommended that affiliation ids are set for new users as soon as possible after provisioning via `setAffiliationIds(ComponentName, Set)` . Non device owners are not subject to this restriction since all privacy-sensitive events happening outside the managed profile would have been redacted already. Starting from `Build.VERSION_CODES.VANILLA_ICE_CREAM` , after the security logging policy has been set, `PolicyUpdateReceiver.onPolicySetResult(Context,String,Bundle,TargetUser,PolicyUpdateResult)` will notify the admin on whether the policy was successfully set or not. This callback will contain:

- The policy identifier `DevicePolicyIdentifiers.SECURITY_LOGGING_POLICY`
- The `TargetUser` that this policy relates to
- The `PolicyUpdateResult` , which will be `PolicyUpdateResult.RESULT_POLICY_SET` if the policy was successfully set or the reason the policy failed to be set e.g. `PolicyUpdateResult.RESULT_FAILURE_CONFLICTING_ADMIN_POLICY`)

If there has been a change to the policy, `PolicyUpdateReceiver.onPolicyChanged(Context,String,Bundle,TargetUser,PolicyUpdateResult)` will notify the admin of this change. This callback will contain the same parameters as PolicyUpdateReceiver#onPolicySetResult and the `PolicyUpdateResult` will contain the reason why the policy changed.

Parameters	
`admin`	`ComponentName` : Which device admin this request is associated with, or `null` if called by a delegated app.
`enabled`	`boolean` : whether security logging should be enabled or not.

Throws	
`SecurityException`	if the caller is not permitted to control security logging.

setShortSupportMessage

```
public void setShortSupportMessage (ComponentName admin,
                CharSequence message)
```

Called by a device admin to set the short support message. This will be displayed to the user in settings screens where functionality has been disabled by the admin. The message should be limited to a short statement such as "This setting is disabled by your administrator. Contact someone@example.com for support." If the message is longer than 200 characters it may be truncated.

If the short support message needs to be localized, it is the responsibility of the `DeviceAdminReceiver` to listen to the `Intent.ACTION_LOCALE_CHANGED` broadcast and set a new version of this string accordingly.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null` .
`message`	`CharSequence` : Short message to be displayed to the user in settings or null to clear the existing message.

Throws	
`SecurityException`	if `admin` is not an active administrator.

setStartUserSessionMessage

```
public void setStartUserSessionMessage (ComponentName admin,
                CharSequence startUserSessionMessage)
```

Called by a device owner to specify the user session start message. This may be displayed during a user switch.

The message should be limited to a short statement or it may be truncated.

If the message needs to be localized, it is the responsibility of the `DeviceAdminReceiver` to listen to the `Intent.ACTION_LOCALE_CHANGED` broadcast and set a new version of this message accordingly.

Parameters	
`admin`	`ComponentName` : which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
`startUserSessionMessage`	`CharSequence` : message for starting user session, or `null` to use system default message.

Throws	
`SecurityException`	if `admin` is not a device owner.

setStatusBarDisabled

```
public boolean setStatusBarDisabled (ComponentName admin,
                boolean disabled)
```

Called by device owner or profile owner of secondary users that is affiliated with the device to disable the status bar. Disabling the status bar blocks notifications and quick settings.

Note: This method has no effect for LockTask mode. The behavior of the status bar in LockTask mode can be configured with `setLockTaskFeatures(ComponentName,int)` . Calls to this method when the device is in LockTask mode will be registered, but will only take effect when the device leaves LockTask mode.

This policy does not have any effect while on the lock screen, where the status bar will not be disabled. Using LockTask instead of this method is recommended.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null` .
`disabled`	`boolean` : `true` disables the status bar, `false` reenables it.

Returns	
`boolean`	`false` if attempting to disable the status bar failed. `true` otherwise.

Throws	
`SecurityException`	if `admin` is not the device owner, or a profile owner of secondary user that is affiliated with the device.

setStorageEncryption

```
public int setStorageEncryption (ComponentName admin,
                boolean encrypt)
```

This method was deprecated in API level 30.
This method does not actually modify the storage encryption of the device. It has never affected the encryption status of a device. Called by an application that is administering the device to request that the storage system be encrypted. Does nothing if the caller is on a secondary user or a managed profile.

When multiple device administrators attempt to control device encryption, the most secure, supported setting will always be used. If any device administrator requests device encryption, it will be enabled; Conversely, if a device administrator attempts to disable device encryption while another device administrator has enabled it, the call to disable will fail (most commonly returning `ENCRYPTION_STATUS_ACTIVE`).

This policy controls encryption of the secure (application data) storage area. Data written to other storage areas may or may not be encrypted, and this policy does not require or control the encryption of any other storage areas. There is one exception: If `Environment.isExternalStorageEmulated()` is `true` , then the directory returned by `Environment.getExternalStorageDirectory()` must be written to disk within the encrypted storage area.

Important Note: On some devices, it is possible to encrypt storage without requiring the user to create a device PIN or Password. In this case, the storage is encrypted, but the encryption key may not be fully secured. For maximum security, the administrator should also require (and check for) a pattern, PIN, or password.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .

`encrypt`	`boolean` : true to request encryption, false to release any previous request
Returns	
`int`	the new total request status (for all active admins), or `DevicePolicyManager.ENCRYPTION_STATUS_UNSUPPORTED` if called for a non-system user. Will be one of `ENCRYPTION_STATUS_UNSUPPORTED` , `ENCRYPTION_STATUS_INACTIVE` , or `ENCRYPTION_STATUS_ACTIVE` . This is the value of the requests; use `getStorageEncryptionStatus()` to query the actual device state.
Throws	
`SecurityException`	if `admin` is not an active administrator or does not use `DeviceAdminInfo.USES_ENCRYPTED_STORAGE`

setSystemSetting

```
public void setSystemSetting (ComponentName admin,
                String setting,
                String value)
```

Called by a device or profile owner to update `Settings.System` settings. Validation that the value of the setting is in the correct form for the setting type should be performed by the caller.

The settings that can be updated by a device owner or profile owner of secondary user with this method are:

- `Settings.System.SCREEN_BRIGHTNESS`
- `Settings.System.SCREEN_BRIGHTNESS_MODE`
- `Settings.System.SCREEN_OFF_TIMEOUT`

Starting from Android `Build.VERSION_CODES.VANILLA_ICE_CREAM` , a profile owner on an organization-owned device can call this method on the parent `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` to set system settings on the parent user.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
`setting`	`String` : The name of the setting to update. This value cannot be `null` . Value is one of the following: • `Settings.System.SCREEN_BRIGHTNESS_MODE` • `Settings.System.SCREEN_BRIGHTNESS` • `Settings.System.SCREEN_OFF_TIMEOUT`
`value`	`String` : The value to update the setting to.

Throws	
SecurityException	if admin is not a device or profile owner.

setSystemUpdatePolicy

```
public void setSystemUpdatePolicy (ComponentName admin,
                SystemUpdatePolicy policy)
```

Called by device owners or profile owners of an organization-owned managed profile to set a local system update policy. When a new policy is set, ACTION_SYSTEM_UPDATE_POLICY_CHANGED is broadcast.

If the supplied system update policy has freeze periods set but the freeze periods do not meet 90-day maximum length or 60-day minimum separation requirement set out in SystemUpdatePolicy.setFreezePeriods , SystemUpdatePolicy.ValidationFailedException will the thrown. Note that the system keeps a record of freeze periods the device experienced previously, and combines them with the new freeze periods to be set when checking the maximum freeze length and minimum freeze separation constraints. As a result, freeze periods that passed validation during SystemUpdatePolicy.setFreezePeriods might fail the additional checks here due to the freeze period history. If this is causing issues during development, adb shell dpm clear-freeze-period-record can be used to clear the record.

Parameters	
admin	ComponentName : Which DeviceAdminReceiver this request is associated with. All components in the package can set system update policies and the most recent policy takes effect. This should be null if the caller is not a device admin.
policy	SystemUpdatePolicy : the new policy, or null to clear the current policy.

Throws	
SystemUpdatePolicy.ValidationFailedException	if the policy's freeze period does not meet the requirement.
IllegalArgumentException	if the policy type or maintenance window is not valid.
SecurityException	if admin is not a device owner or a profile owner of an organization-owned managed profile, or the caller is not permitted to set this policy

setTime

```
public boolean setTime (ComponentName admin,
                long millis)
```

Called by a device owner or a profile owner of an organization-owned managed profile to set the system wall clock time. This only takes effect if called when Settings.Global.AUTO_TIME is 0, otherwise false will be returned.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null` .
`millis`	`long` : time in milliseconds since the Epoch

Returns	
`boolean`	`true` if set time succeeded, `false` otherwise.

Throws	
`SecurityException`	if `admin` is not a device owner or a profile owner of an organization-owned managed profile.

setTimeZone

```
public boolean setTimeZone (ComponentName admin,
                String timeZone)
```

Called by a device owner or a profile owner of an organization-owned managed profile to set the system's persistent default time zone. This only takes effect if called when `Settings.Global.AUTO_TIME_ZONE` is 0, otherwise `false` will be returned.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null` .
`timeZone`	`String` : one of the Olson ids from the list returned by `TimeZone.getAvailableIDs()`

Returns	
`boolean`	`true` if set timezone succeeded, `false` otherwise.

Throws	
`SecurityException`	if `admin` is not a device owner or a profile owner of an organization-owned managed profile.

setTrustAgentConfiguration

```
public void setTrustAgentConfiguration (ComponentName admin,
                ComponentName target,
```

```
                PersistableBundle configuration)
```

Sets a list of configuration features to enable for a trust agent component. This is meant to be used in conjunction with `KEYGUARD_DISABLE_TRUST_AGENTS`, which disables all trust agents but those enabled by this function call. If flag `KEYGUARD_DISABLE_TRUST_AGENTS` is not set, then this call has no effect.

For any specific trust agent, whether it is disabled or not depends on the aggregated state of each admin's `KEYGUARD_DISABLE_TRUST_AGENTS` setting and its trust agent configuration as set by this function call. In particular: if any admin sets `KEYGUARD_DISABLE_TRUST_AGENTS` and does not additionally set any trust agent configuration, the trust agent is disabled completely. Otherwise, the trust agent will receive the list of configurations from all admins who set `KEYGUARD_DISABLE_TRUST_AGENTS` and aggregate the configurations to determine its behavior. The exact meaning of aggregation is trust-agent-specific.

A calling device admin must have requested `DeviceAdminInfo.USES_POLICY_DISABLE_KEYGUARD_FEATURES` to be able to call this method; if not, a security exception will be thrown.

This method can be called on the `DevicePolicyManager` instance returned by `getParentProfileInstance(ComponentName)` in order to set the configuration for the parent profile.

On devices not supporting `PackageManager.FEATURE_SECURE_LOCK_SCREEN` feature, calling this method has no effect - no trust agent configuration will be set.
Requires the `PackageManager#FEATURE_SECURE_LOCK_SCREEN` feature which can be detected using `PackageManager.hasSystemFeature(String)`.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null`.
`target`	`ComponentName`: Component name of the agent to be configured. This value cannot be `null`.
`configuration`	`PersistableBundle`: Trust-agent-specific feature configuration bundle. Please consult documentation of the specific trust agent to determine the interpretation of this bundle.
Throws	
`SecurityException`	if `admin` is not an active administrator or does not use `DeviceAdminInfo.USES_POLICY_DISABLE_KEYGUARD_FEATURES`

setUninstallBlocked

```
public void setUninstallBlocked (ComponentName admin,
                String packageName,
                boolean uninstallBlocked)
```

Change whether a user can uninstall a package. This function can be called by a device owner, profile owner, or by a delegate given the `DELEGATION_BLOCK_UNINSTALL` scope via `setDelegatedScopes(ComponentName, String, List)` or holders of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_APPS_CONTROL`.

Starting from `Build.VERSION_CODES.UPSIDE_DOWN_CAKE`, after the set uninstall blocked policy has been set, `PolicyUpdateReceiver.onPolicySetResult(Context,String,Bundle,TargetUser,PolicyUpdateResult)` will notify the admin on whether the policy was successfully set or not. This callback will contain:

- The policy identifier `DevicePolicyIdentifiers.PACKAGE_UNINSTALL_BLOCKED_POLICY`
- The additional policy params bundle, which contains `PolicyUpdateReceiver.EXTRA_PACKAGE_NAME` the package name the policy applies to
- The `TargetUser` that this policy relates to
- The `PolicyUpdateResult`, which will be `PolicyUpdateResult.RESULT_POLICY_SET` if the policy was successfully set or the reason the policy failed to be set (e.g. `PolicyUpdateResult.RESULT_FAILURE_CONFLICTING_ADMIN_POLICY`)

If there has been a change to the policy, `PolicyUpdateReceiver.onPolicyChanged(Context,String,Bundle,TargetUser,PolicyUpdateResult)` will notify the admin of this change. This callback will contain the same parameters as PolicyUpdateReceiver#onPolicySetResult and the `PolicyUpdateResult` will contain the reason why the policy changed.

Parameters	
`admin`	`ComponentName`: Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null`.
`packageName`	`String`: package to change.
`uninstallBlocked`	`boolean`: true if the user shouldn't be able to uninstall the package.
Throws	
`SecurityException`	if `admin` is not a device or profile owner or holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_APPS_CONTROL`.

setUsbDataSignalingEnabled

```
public void setUsbDataSignalingEnabled (boolean enabled)
```

Called by a device owner or profile owner of an organization-owned managed profile to enable or disable USB data signaling for the device. When disabled, USB data connections (except from charging functions) are prohibited.

This API is not supported on all devices, the caller should call `canUsbDataSignalingBeDisabled()` to check whether enabling or disabling USB data signaling is supported on the device. Starting from Android 15, after the USB data signaling policy has been set,

`PolicyUpdateReceiver.onPolicySetResult(Context,String,Bundle,TargetUser,PolicyUpdateResult)` will notify the admin on whether the policy was successfully set or not. This callback will contain:

- The `TargetUser` that this policy relates to
- The `PolicyUpdateResult`, which will be `PolicyUpdateResult.RESULT_POLICY_SET` if the policy was successfully set or the reason the policy failed to be set e.g. `PolicyUpdateResult.RESULT_FAILURE_CONFLICTING_ADMIN_POLICY`)

If there has been a change to the policy, `PolicyUpdateReceiver.onPolicyChanged(Context,String,Bundle,TargetUser,PolicyUpdateResult)` will notify the admin of this change. This callback will contain the same parameters as PolicyUpdateReceiver#onPolicySetResult and the `PolicyUpdateResult` will contain the reason why the policy changed.

Parameters	
`enabled`	`boolean` : whether USB data signaling should be enabled or not.
Throws	
`IllegalStateException`	if disabling USB data signaling is not supported or if USB data signaling fails to be enabled/disabled.
`SecurityException`	if the caller is not permitted to set this policy

setUserControlDisabledPackages

```
public void setUserControlDisabledPackages (ComponentName admin,
                List<String> packages)
```

Called by a device owner or a profile owner or holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_APPS_CONTROL` to disable user control over apps. User will not be able to clear app data or force-stop packages. When called by a device owner, applies to all users on the device. Packages with user control disabled are exempted from App Standby Buckets.

Starting from `Build.VERSION_CODES.UPSIDE_DOWN_CAKE`, after the user control disabled packages policy has been set, `PolicyUpdateReceiver.onPolicySetResult(Context,String,Bundle,TargetUser,PolicyUpdateResult)` will notify the admin on whether the policy was successfully set or not. This callback will contain:

- The policy identifier `DevicePolicyIdentifiers.USER_CONTROL_DISABLED_PACKAGES_POLICY`
- The `TargetUser` that this policy relates to
- The `PolicyUpdateResult`, which will be `PolicyUpdateResult.RESULT_POLICY_SET` if the policy was successfully set or the reason the policy failed to be set (e.g. `PolicyUpdateResult.RESULT_FAILURE_CONFLICTING_ADMIN_POLICY`)

If there has been a change to the policy, `PolicyUpdateReceiver.onPolicyChanged(Context,String,Bundle,TargetUser,PolicyUpdateResult)` will notify the

admin of this change. This callback will contain the same parameters as PolicyUpdateReceiver#onPolicySetResult and the `PolicyUpdateResult` will contain the reason why the policy changed.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. Null if the caller is not a device admin. This value may be `null` .
`packages`	`List` : The package names for the apps. This value cannot be `null` .
Throws	
`SecurityException`	if `admin` is not a device owner or a profile owner or holder of the permission `Manifest.permission.MANAGE_DEVICE_POLICY_APPS_CONTROL` .

setUserIcon

```
public void setUserIcon (ComponentName admin,
                Bitmap icon)
```

Called by profile or device owners to set the user's photo.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
`icon`	`Bitmap` : the bitmap to set as the photo.
Throws	
`SecurityException`	if `admin` is not a device or profile owner.

setWifiSsidPolicy

```
public void setWifiSsidPolicy (WifiSsidPolicy policy)
```

Called by device owner or profile owner of an organization-owned managed profile to specify the Wi-Fi SSID policy (`WifiSsidPolicy`). Wi-Fi SSID policy specifies the SSID restriction the network must satisfy in order to be eligible for a connection. Providing a null policy results in the deactivation of the SSID restriction

Parameters	
`policy`	`WifiSsidPolicy` : Wi-Fi SSID policy. This value may be `null` .

Throws	
SecurityException	if the caller is not permitted to manage wifi policy

startUserInBackground

```
public int startUserInBackground (ComponentName admin,
                UserHandle userHandle)
```

Called by a device owner to start the specified secondary user in background.

Parameters	
admin	ComponentName : Which DeviceAdminReceiver this request is associated with. This value cannot be null .
userHandle	UserHandle : the user to be started in background. This value cannot be null .

Returns	
int	one of the following result codes: UserManager.USER_OPERATION_ERROR_UNKNOWN , UserManager.USER_OPERATION_SUCCESS , UserManager.USER_OPERATION_ERROR_MANAGED_PROFILE , UserManager.USER_OPERATION_ERROR_MAX_RUNNING_USERS ,. Value is one of the following: • UserManager.USER_OPERATION_SUCCESS • UserManager.USER_OPERATION_ERROR_UNKNOWN • UserManager.USER_OPERATION_ERROR_MANAGED_PROFILE • UserManager.USER_OPERATION_ERROR_MAX_RUNNING_USERS • UserManager.USER_OPERATION_ERROR_CURRENT_USER • UserManager.USER_OPERATION_ERROR_LOW_STORAGE • UserManager.USER_OPERATION_ERROR_MAX_USERS

Throws	
SecurityException	if admin is not a device owner.

stopUser

```
public int stopUser (ComponentName admin,
                UserHandle userHandle)
```

Called by a device owner to stop the specified secondary user.

Parameters

`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
`userHandle`	`UserHandle` : the user to be stopped. This value cannot be `null` .

Returns

`int`	one of the following result codes: `UserManager.USER_OPERATION_ERROR_UNKNOWN` , `UserManager.USER_OPERATION_SUCCESS` , `UserManager.USER_OPERATION_ERROR_MANAGED_PROFILE` , `UserManager.USER_OPERATION_ERROR_CURRENT_USER` Value is one of the following: • `UserManager.USER_OPERATION_SUCCESS` • `UserManager.USER_OPERATION_ERROR_UNKNOWN` • `UserManager.USER_OPERATION_ERROR_MANAGED_PROFILE` • `UserManager.USER_OPERATION_ERROR_MAX_RUNNING_USERS` • `UserManager.USER_OPERATION_ERROR_CURRENT_USER` • `UserManager.USER_OPERATION_ERROR_LOW_STORAGE` • `UserManager.USER_OPERATION_ERROR_MAX_USERS`

Throws

`SecurityException`	if `admin` is not a device owner.

switchUser

```
public boolean switchUser (ComponentName admin,
                UserHandle userHandle)
```

Called by a device owner to switch the specified secondary user to the foreground.

Parameters

`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
`userHandle`	`UserHandle` : the user to switch to; null will switch to primary.

Returns

`boolean`	`true` if the switch was successful, `false` otherwise.

Throws

`SecurityException`	if `admin` is not a device owner.

transferOwnership

```
public void transferOwnership (ComponentName admin,
                ComponentName target,
                PersistableBundle bundle)
```

Changes the current administrator to another one. All policies from the current administrator are migrated to the new administrator. The whole operation is atomic - the transfer is either complete or not done at all.

Depending on the current administrator (device owner, profile owner), you have the following expected behaviour:

- A device owner can only be transferred to a new device owner
- A profile owner can only be transferred to a new profile owner

Use the `bundle` parameter to pass data to the new administrator. The data will be received in the `DeviceAdminReceiver.onTransferOwnershipComplete(Context,PersistableBundle)` callback of the new administrator.

The transfer has failed if the original administrator is still the corresponding owner after calling this method.

The incoming target administrator must have the `<support-transfer-ownership />` tag inside the `<device-admin> </device-admin>` tags in the xml file referenced by `DeviceAdminReceiver.DEVICE_ADMIN_META_DATA`. Otherwise an `IllegalArgumentException` will be thrown.

Parameters	
`admin`	`ComponentName` : which `DeviceAdminReceiver` this request is associated with. This value cannot be `null`.
`target`	`ComponentName` : which `DeviceAdminReceiver` we want the new administrator to be. This value cannot be `null`.
`bundle`	`PersistableBundle` : data to be sent to the new administrator. This value may be `null`.

Throws	
`IllegalArgumentException`	if `admin` or `target` is `null`, they are components in the same package or `target` is not an active admin.
`SecurityException`	if `admin` is not a device owner nor a profile owner.

uninstallAllUserCaCerts

```
public void uninstallAllUserCaCerts (ComponentName admin)
```

Uninstalls all custom trusted CA certificates from the profile. Certificates installed by means other than device policy will also be removed, except for system CA certificates.

Parameters

`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with, or `null` if calling from a delegated certificate installer.
Throws	
`SecurityException`	if `admin` is not `null` and not a device or profile owner.

uninstallCaCert

```
public void uninstallCaCert (ComponentName admin,
                byte[] certBuffer)
```

Uninstalls the given certificate from trusted user CAs, if present. The caller must be a profile or device owner on that user, or a delegate package given the `DELEGATION_CERT_INSTALL` scope via `setDelegatedScopes(ComponentName, String, List)` ; otherwise a security exception will be thrown.

Parameters	
`admin`	`ComponentName` : Which `DeviceAdminReceiver` this request is associated with, or `null` if calling from a delegated certificate installer.
`certBuffer`	`byte` : encoded form of the certificate to remove.
Throws	
`SecurityException`	if `admin` is not `null` and not a device or profile owner.

updateOverrideApn

```
public boolean updateOverrideApn (ComponentName admin,
                int apnId,
                ApnSetting apnSetting)
```

Called by device owner or managed profile owner to update an override APN.

This method may returns `false` if there is no override APN with the given `apnId` .

This method may also returns `false` if `apnSetting` conflicts with an existing override APN. Update the existing conflicted APN instead.

See `addOverrideApn(ComponentName, ApnSetting)` for the definition of conflict.

Before Android version `Build.VERSION_CODES.TIRAMISU` : Only device owners can update APNs.

Starting from Android version `Build.VERSION_CODES.TIRAMISU` : Both device owners and managed profile owners can update enterprise APNs (`ApnSetting.TYPE_ENTERPRISE`), while only device owners can update other type of APNs.

Parameters	
`admin`	`ComponentName` : which `DeviceAdminReceiver` this request is associated with. This value cannot be `null` .
`apnId`	`int` : the `id` of the override APN to update
`apnSetting`	`ApnSetting` : the override APN to update. This value cannot be `null` .

Returns	
`boolean`	`true` if the required override APN is successfully updated, `false` otherwise.

Throws	
`Security Exception`	If request is for enterprise APN `admin` is either device owner or profile owner and in all other types of APN if `admin` is not a device owner.

wipeData

```
public void wipeData (int flags,
                CharSequence reason)
```

Ask that all user data be wiped.

If called as a secondary user or managed profile, the user itself and its associated user data will be wiped. In particular, If the caller is a profile owner of an organization-owned managed profile, calling this method will relinquish the device for personal use, removing the managed profile and all policies set by the profile owner.

Calling this method from the primary user will only work if the calling app is targeting SDK level `Build.VERSION_CODES.TIRAMISU` or below, in which case it will cause the device to reboot, erasing all device data - including all the secondary users and their data - while booting up. If an app targeting SDK level `Build.VERSION_CODES.UPSIDE_DOWN_CAKE` and above is calling this method from the primary user or last full user, `IllegalStateException` will be thrown.

If an app wants to wipe the entire device irrespective of which user they are from, they should use `wipeDevice(int)` instead.

Parameters	
`flags`	`int` : Bit mask of additional options: currently supported flags are `WIPE_EXTERNAL_STORAGE` and `WIPE_RESET_PROTECTION_DATA` .
`reason`	`CharSequence` : a string that contains the reason for wiping data, which can be presented to the user. This value cannot be `null` .

Throws

`IllegalArgumentException`	if the input reason string is null or empty, or if `WIPE_SILENTLY` is set.
`IllegalStateException`	if called on last full-user or system-user
`SecurityException`	if the calling application does not own an active administrator that uses `DeviceAdminInfo.USES_POLICY_WIPE_DATA` and is not granted the `Manifest.permission.MASTER_CLEAR` or `Manifest.permission.MANAGE_DEVICE_POLICY_WIPE_DATA` permissions.

wipeData

```
public void wipeData (int flags)
```

See `wipeData(int,CharSequence)`

Parameters	
`flags`	`int` : Bit mask of additional options: currently supported flags are `WIPE_EXTERNAL_STORAGE` , `WIPE_RESET_PROTECTION_DATA` and `WIPE_SILENTLY` .
Throws	
`IllegalStateException`	if called on last full-user or system-user
`SecurityException`	if the calling application does not own an active administrator that uses `DeviceAdminInfo.USES_POLICY_WIPE_DATA` and is not granted the `Manifest.permission.MASTER_CLEAR` or `Manifest.permission.MANAGE_DEVICE_POLICY_WIPE_DATA` permissions.

wipeDevice

```
public void wipeDevice (int flags)
```

Ask that the device be wiped and factory reset.

The calling Device Owner or Organization Owned Profile Owner must have requested `DeviceAdminInfo.USES_POLICY_WIPE_DATA` to be able to call this method; if it has not, a security exception will be thrown.

Parameters	
`flags`	`int` : Bit mask of additional options: currently supported flags are `WIPE_EXTERNAL_STORAGE` , `WIPE_RESET_PROTECTION_DATA` , `WIPE_EUICC` and `WIPE_SILENTLY` .
Throws	

`SecurityException`	if the calling application does not own an active administrator that uses `DeviceAdminInfo.USES_POLICY_WIPE_DATA` and is not granted the `Manifest.permission.MASTER_CLEAR` or both the `Manifest.permission.MANAGE_DEVICE_POLICY_WIPE_DATA` and `Manifest.permission.MANAGE_DEVICE_POLICY_ACROSS_USERS` permissions.

Source: https://developer.android.com/reference/android/app/admin/DevicePolicyManager#setPermittedCrossProfileNotificationListeners(android.content.ComponentName,%20java.util.List%3Cjava.lang.String%3E)